UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

MEMS Sensor Data Acquisition Framework Based on Arduino Platform

Trabajo de Fin de Grado

Presentado en la

Escola Tècnica Superior d'Enginyeria de Telecomunicacions de Barcelona

Universitat Politècnica de Catalunya

por

Eusebi Forrellad Munné

En cumplimiento parcial

de los requisitos para el Grado en

Ingeniería de Sistemas Electrónicos

Directores: Josep Maria Sánchez Chiva y Jordi Madrenas

Barcelona, Junio 2017

Abstract

Research groups that work with sensors, for instance AHA (Advanced Hardware Architectures) group where this project has been developed, frequently need to design experimental setups. For each experiment, data acquisition, processing, visualization and storage has to be designed, which is a time consuming task.

In this project, the data acquisition, processing, visualization and its associated storage was implemented using the *Instrumentino* program, an open source modular graphical interface framework for controlling experiments and the experimental measurements based on Arduino. The system has been customized for a set of printed circuits boards designed to test a Lorentz force based magnetic sensor, a project that is currently being developed by the *AHA* research group.

Besides developing a practical and flexible environment to verify the above mentioned project, this Bachelor Thesis targets to be a useful tool for future projects of the research group.

Resum

Els grups d'investigació que treballen amb sensors, com el grup d'investigació AHA (*Advanced Hardware Architectures*) en el qual s'ha realitzat aquest Treball de Fi de Grau (TFG), sovint han de realitzar proves i mesures experimentals durant el desenvolupament dels seus projectes. En cada projecte s'escull la forma en la que es duran a terme les diferents proves: com s'adquireixen les dades, es processen, es visualitzen i s'emmagatzemen. Aquestes tasques acaben representant una part important de la dedicació en els projectes d'investigació.

En aquest Treball de Fi de Grau s'ha desenvolupat el sistema d'adquisició de dades, el processat, la visualització dels resultats i l'enmagatzematge d'aquests en un fitxer, tot això mitjançant el programa Instrumentino, un *framework* amb una interfície gràfica modular de codi obert per controlar experiments i mesures experimentals basades en Arduino. El sistema s'ha personalitzat per a una placa de circuit imprès realitzada per testejar un sensor magnètic basat en la força de Lorentz, un projecte que actualment s'està realitzant al grup AHA.

A part de desenvolupar un entorn pràctic i flexible per verificar el projecte anteriorment mencionat, aquest TFG vol ser una eina útil per futurs a projectes del grup d'investigació.

Resumen

Los grupos de investigación que trabajan con sensores, como el grupo de investigación AHA (*Advanced Hardware Architectures*) en cuyo ámbito se ha realizado este Trabajo de Fin de Grado (TFG), a menudo tienen que realizar pruebas y medidas experimentales durante el desarrollo de sus proyectos. En cada proyecto se escoge la forma en que se desarrollan las diferentes pruebas, como se adquieren los datos, se procesan, se visualizan y se almacenan. Estas tareas acaban representando una parte importante de dedicación en los proyectos de investigación.

En este TFG se ha desarrollado el sistema de adquisición de datos, el procesado, la visualización de los resultados y el almacenamiento de los mismos en un fichero, todo esto mediante el programa Instrumentino, un *framework* con interfaz gráfica modular de código abierto para el control de experimentos y medidas experimentales basado en Arduino. El sistema se ha personalizado para una placa de circuito impreso realizada para caracterizar un sensor magnético basado en la fuerza de Lorentz, un proyecto que actualmente se está realizando en el grupo AHA.

Aparte de desarrollar un entorno práctico y flexible para verificar el proyecto anteriormente mencionado, este TFG quiere ser una herramienta útil para futuros proyectos del grupo de investigación.

Agradecimientos

En primer lugar me gustaría agradecer a Josep Maria Sánchez Chiva todo su trabajo y dedicación durante todo el proyecto, tanto en la parte del *hardware* como la del *software*. Este proyecto no podría haberse realizado sin su ayuda.

También quiero agradecer a Jordi Madrenas Boadas su continuo soporte y asistencia a lo largo del proyecto.

Historial de revisiones y registro de aprobación

Revision	Date	Purpose
0	21/06/2017	Document creation
1	22/06/2017	Document revision
2	26/06/2017	Document revision
3	29/06/2017	Document revision

LISTA DE DISTRIBUCIÓN DE DOCUMENTOS

Name	e-mail
Eusebi Forrellad	actip3@gmail.com
Jordi Madrenas	jordi.madrenas@upc.edu
Josep Maria Sánchez Chiva	jose.maria.sanchez.chiva@upc.edu

Written by:		Reviewed and approved by:	
Date	21/06/2017	Date	29/06/2017
Name	Eusebi Forrellad	Name	Josep Maria Sánchez Chiva
Position	Project Author	Position	Project Supervisor

Contenidos

1. Introducción

1.1. Introducción al proyecto

Este proyecto está relacionado con dos tesis doctorales realizadas en el mismo grupo de investigación AHA, cuyo principal tema de investigación es el desarrollo de sensores innovadores *MicroElectroMechanical Systems* (MEMS) para una variedad de magnitudes físicas.
La primera de ellas trata sobre el diseño de sensores magnéticos MEMS basados en la fuerza de Lorentz y la segunda con la electrónica de acondicionamiento del mismo, ambas se están llevando a cabo actualmente.
Anteriormente, para la lectura de datos se utilizaba un programa personalizado implementado en C, sin interfaz gráfica. Una solución poco robusta y que en caso de mejorarla solo serviría para este proyecto.
Tras barajar distintas opciones con el objetivo de desarrollar un entorno que también fuera adaptable para futuros proyectos, se decidió optar por Instrumentino [1,2]. Instrumentino es un *framework* con interfaz gráfica modular de código abierto para controlar experimentos y medidas experimentales basados en Arduino.
También permite la creación de programas de control para sistemas complejos con un esfuerzo de programación menor que el de desarrollar un programa desde el inicio. Mediante la escritura de un único archivo de código, se genera una interfaz de usuario personalizada, que permite el funcionamiento automático de secuencias de operaciones elaboradas y la observación de datos experimentales adquiridos en tiempo real.
Dicho entorno contiene Controlino, un programa (llamado *Sketch* en el entorno Arduino) que proporciona una forma sencilla de controlar y monitorizar Arduino a través del cable USB.

1.2. Objetivos

Los objetivos de este proyecto se pueden dividir en:

1.- Implementar los cambios y funciones necesarias en el Controlino para la adquisición de datos.
2.- Implementar los cambios y funciones necesarias en el Controlino para la comunicación con los diferentes componentes de la placa.
3.- Implementar los cambios y funciones necesarias en el Instrumentino para el funcionamiento de los objetivos 1 y 2.
4.- Guardar los datos correctamente en un fichero csv.
5.- Crear un script que genere un entorno de trabajo en la plataforma Instrumentino que permita mostrar en la interfaz gráfica los datos obtenidos en tiempo real.
6.- Colgar en la plataforma donde se encuentra Instrumentino las mejoras realizadas con Instrumentino para que otros usuarios también puedan beneficiarse.

1.3. Enfoque del problema

En primer lugar, para poder visualizar los resultados de los experimentos realizados, se utiliza la herramienta GUI (Graphical User Interface). Las prestaciones que nos ofrece este programa informático son las de representarnos la información obtenida del experimento por medio de imágenes y objetos gráficos. El objetivo primordial de esta interfaz es ofrecer un entorno visual básico en el que establecer la comunicación con el sistema operativo de un dispositivo o de un ordenador.

Cabe destacar que en el desarrollo de toda investigación se presentan numerosas trabas de las que se requiere largo tiempo de estudio. Sin embargo, no siempre se dispone de tiempo, y las habilidades del que desempeña el proyecto no alcanzan a resolver los problemas. Este proyecto es un ejemplo de muchos en los que ciertos apartados de electrónica se han llevado a cabo sin gran dificultad, como son el mecanizado, la instrumentación y la electrónica, y otros han supuesto un obstáculo, como en algunas partes de la programación.

Se pueden implementar una GUI basada en MATLAB para Arduino [3], como también es posible usar entornos de programación visual para la creaciones de GUI's personalizadas tales como LabView [4] o Visual Basic [5]. Como agravante de estas, sus licencias son sumamente caras y demasiado complejas de usar.

En estos últimos años ha surgido una nueva tendencia entre los usuarios que necesitan utilizar GUI's en sus experimentos, se trata del entorno del hardware de código abierto.

Para realizar una interfaz gráfica con estos dispositivos existen alternativas también de código abierto, como Guino [6], este programa permite al usuario crear una GUI personalizada con capacidades de control y una gráfica, el programa comunica el PC con el arduino, sin embargo la personalización de la GUI la realiza el programa del Arduino y esto afecta al rendimiento de los experimentos debido al limitado poder de procesamiento de los arduinos.

2. Hardware

El hardware utilizado durante el proyecto está centrado en el sistema de PCBs de la Fig. 1.

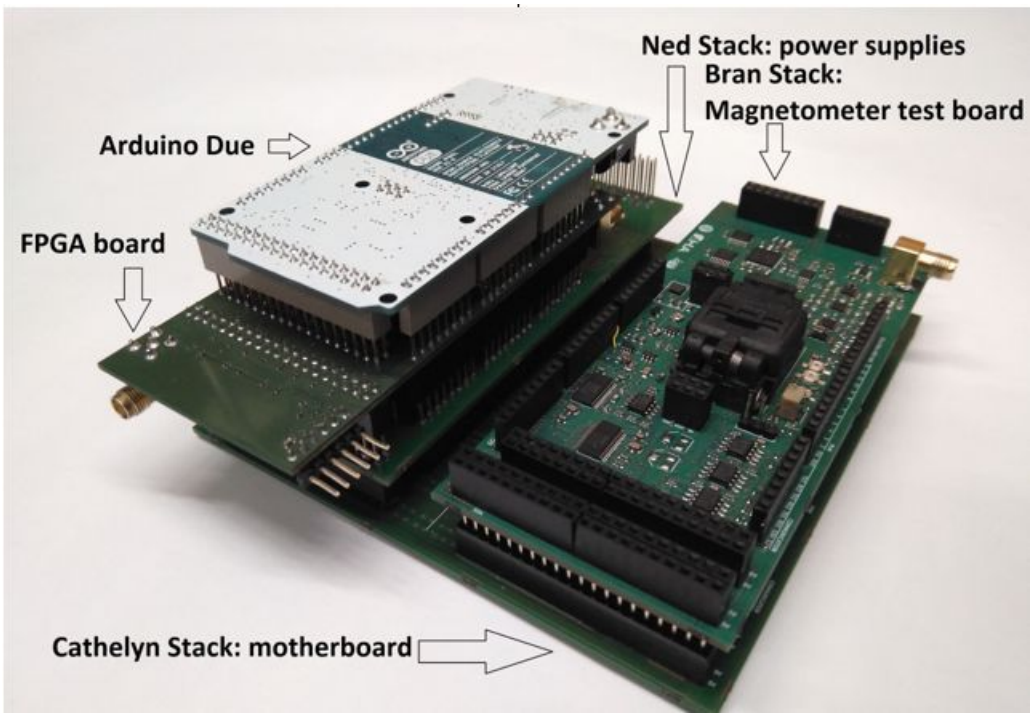

Figura 1: Imagen de la plataforma hardware

La plataforma hardware consiste en un *stack* de placas, cada una de las cuales cumple con una función concreta. La placa madre sirve de soporte y conexión para el resto de placas. Por un lado, se encuentra una placa con los convertidores dc-dc, la placa con la FPGA y encima el Arduino. Por otro lado, se encuentra la placa de acondicionamiento de señal para el dispositivo en estudio DUT(*device under test*), así como la electrónica necesaria para la adquisición de la señal.

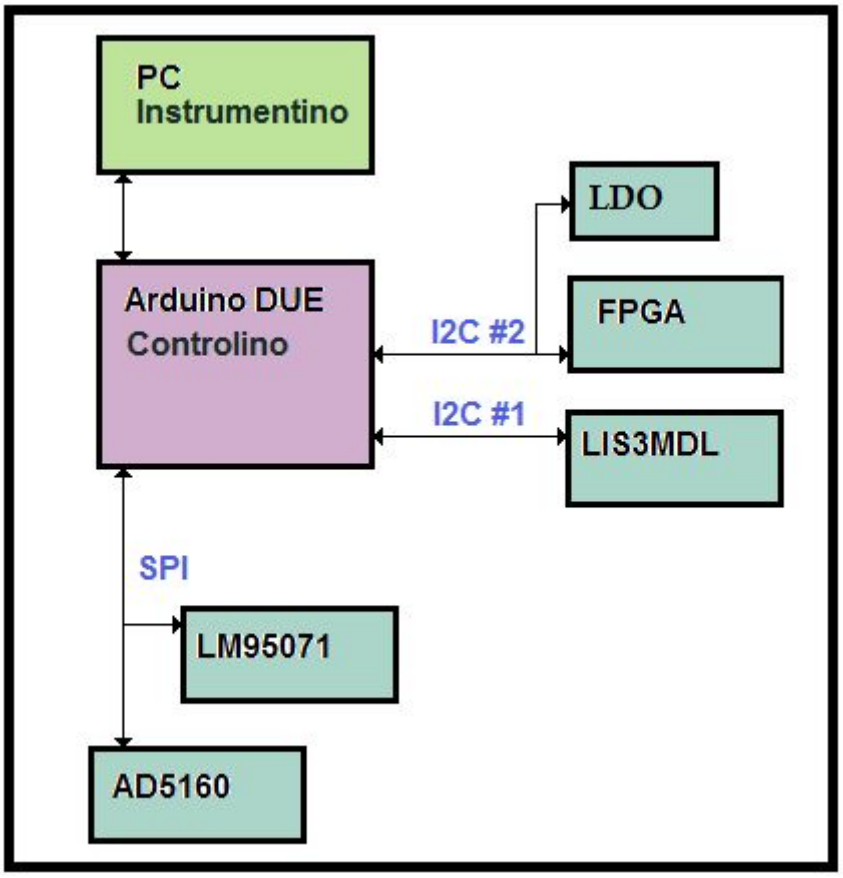

Figura 2: Diagrama de bloques de la parte digital de la placa base

En la Fig. 2 se puede observar el diagrama de bloques de la parte digital de las placas mostrada anteriormente. En primer lugar se encuentra el PC con el programa Instrumentino, este se comunica mediante el puerto USB con el *Sketch* cargado en el Arduino Due llamado Controlino.

A continuación el Arduino Due es el encargado de comunicarse con los demás componentes de la placa base. Mediante el protocolo SPI se comunica con el sensor de temperatura LM95071 y con el potenciómetro digital AD5160, el cual funciona como divisor de tensión para controlar la tensión dc a la entrada del DUT.

El arduino Due dispone de dos interfaces I2C. El sensor de referencia LIS3MDL, usa una de ellas, mientras que el segundo I2C se utiliza para comunicarse con la FPGA y para configurar el regulador LDO (*Low drop out*) que regula las tensiones de alimentación de la placa donde se encuentra la FPGA.

Finalmente la FPGA está comunicada con el sensor DUT a través de los datos que recibe del ADC, la FPGA las procesa y las envía al Arduino que recibe la amplitud y

frecuencia del sensor procesados

2.1 Arduino DUE

Arduino es una placa programable con un microcontrolador, normalmente del fabricante Atmel, con un puerto USB para programar y alimentar el equipo y con entradas y salidas digitales y analógicas. También cuenta con un entorno de desarrollo (IDE), con el que se programa cada placa.

En el caso de este proyecto se utiliza la placa Arduino DUE (Fig. 3).

Figura 3: Placa Arduino Due [7]

Esta placa se basa en un microcontrolador ARM CortexM3 de 32 bits, y tiene una velocidad de CPU de 84MHz.
Arduino Due está compuesto por un total de 54 pines de entradas o salidas digitales, 12 de ellas funcionales como PWM, también dispone de 12 entradas analógicas, 4 UARTs, una conexión USB-OTG, 2 DAC, 2TWI y una comunicación SPI .
Al no ser uno de los Arduinos más utilizados y al usar un microcontrolador de 32bits, aún se dispone de poco soporte por parte de la comunidad de Arduino.

2.2 FPGA

La FPGA (*Field Programmable Gate Array*) se encarga de generar señales de control y de realizar procesado de alta velocidad (por ejemplo, filtrado digital).

2.3 Sensor de referencia LIS3MDL

El LIS3MDL [8] es un sensor magnético comercial de tres ejes con una resolución de 16 bits y una escala completa que puede variar entre ± 4 / ± 8 / ± 12 / ± 16 gauss. Este sensor incluye una interfaz de bus serie I2C y la interfaz estándar serie SPI. Se usa como referencia para comparar con el dispositivo a medir.

2.4 Sensor de temperatura LM95071

El LM95071 [9] es un sensor de temperatura digital de alta resolución, compatible para la comunicación mediante SPI. Tiene una resolución de temperatura de señal de 13 bits y una sensibilidad de 0.03125˚C/LSB), en un rango desde -40ºC hasta 150ºC. Se emplea para detectar posibles variaciones de las medidas con la temperatura.

2.5 Acelerómetro LIS331DLH

El LIS331DLH [10] es un acelerómetro lineal de tres ejes, con salida estándar de interfaz serie digital I2C / SPI.
Tiene escalas completas de ± 2g / ± 4g / ± 8g seleccionables por el usuario y es capaz de medir aceleraciones con velocidades de salida de 0,5 Hz a 1 kHz.

La intefaz de este acelerómetro es muy parecida a la del magnetómetro de referencia utilizado en la placa base, ambos están fabricados por la empresa STMicroelectrics, usan la comunicación (I2C) y el banco de registros es también muy similar, por lo que ha sido utilizado para desarrollar la comunicación I2C con el Instrumentino, y así facilitar el desarrollo, al no tener que usar la placa base.

2.6 Potenciómetro AD5160

Los potenciómetros digitales como el AD5160 [11] son útiles cuando se necesita variar la resistencia en un circuito de manera digital en lugar de manualmente.
Los ajustes son controlables a través de una comunicación SPI. En el caso de este modelo la resistencia varía linealmente con respecto al código digital transferido.

3 Comunicaciones

Para el desarrollo del proyecto ha sido necesario utilizar un total de tres tipos diferentes de comunicaciones.

3.1 Comunicación Serie

El puerto Serial es una interfaz que solo requiere dos líneas de señal para enviar y recibir datos, los conectores RX (recepción) y TX (transmisión), los cuales utilizan un protocolo de datos asíncrono.

Los ordenadores disponen de varios puertos de serie. Los más conocidos son el USB (universal serial port) y menos utilizado el RS-232.

3.2 Comunicación SPI

El SPI (*Synchronous Peripheral Interface*) es un protocolo sencillo de comunicación serie síncrona de 4 hilos utilizado para la comunicación serie entre dispositivos. Fue creado por Motorola y adoptado posteriormente por otros fabricantes, como Atmel y Microchip.

Su diseño permite la transmisión de datos a velocidades de 10Mbps y al operar en modo *full duplex*, permite que las señales de datos se transmiten en ambas direcciones simultáneamente.
Los dispositivos SPI se comunican entre sí utilizando un esquema maestro/esclavo, en el que el maestro inicia la trasmisión de los datos.

Descripción de las señales
MOSI (Master-out, slave-in) para la comunicación del maestro al esclavo.
MISO (Master-in, slave-out) para comunicación del esclavo al maestro.
SS (Slave Select) El pin que el maestro usa para habilitar y deshabilitar dispositivos específicos.
SCLK (System Clock) señal de reloj enviada por el maestro.

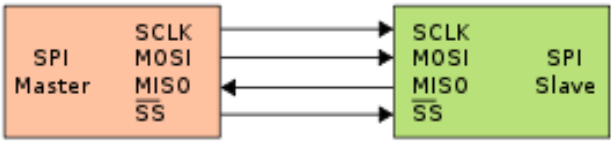

Figura 4: Ejemplo de una comunicación SPI entre un dispositivo ejerciendo de maestro y otro de esclavo [12]

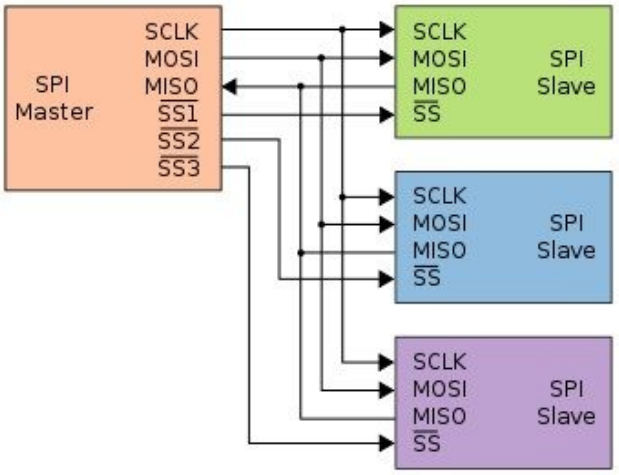

Figura 5: Ejemplo de una comunicación SPI entre un dispositivo ejerciendo de maestro y tres esclavos [13]

En la Fig. 4 se muestra un ejemplo con solamente un dispositivo maestro y un esclavo.

Sin embargo, añadiendo varias líneas SS, se puede implementar una red de varios circuitos SPI, controlados por el mismo dispositivo maestro, como se muestra en la Fig. 5.

3.3 Comunicación I2C

I2C (*inter integrated circuits*), es un tipo de bus que fue diseñado por Philips Semiconductors [x]. El I2C es un bus en el cual se pueden conectar varios dispositivos donde uno realiza la función de maestro y el resto de esclavos.

El bus I2C es un estándar que sólo requiere de dos líneas de señal y un común o masa. Permite el intercambio de información a una velocidad aceptable, de unos 100 kbit/s.

La metodología de comunicación de datos del bus I2C es en serie y sincróna.

Descripción de las señales
SCL (System Clock) es la línea de los pulsos de reloj que sincronizan el sistema.
SDA (System Data) es la línea por la que se mueven los datos entre los dispositivos.

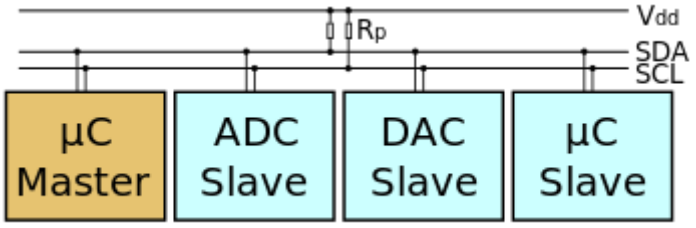

Figura 6: Ejemplo de una comunicación I2C entre un dispositivo ejerciendo de maestro y tres esclavos [14]

En la Fig. 6 se muestra un ejemplo donde un microcontrolador (maestro) se comunica con diversos dispositivos, un ADC, un DAC y otro microcontrolador, cada uno de los dispositivos tiene una dirección que el microcontrolador maestro ha de conocer para la correcta comunicación. También es necesario conectar dos resistencias de polarización, porque las dos líneas del bus deben estar en un nivel lógico alto cuando están inactivas, los valores más comunes son entre 1K5 y 10K.

4 Software

4.1 Instrumentino

4.1.1 Introducción

El software principal del proyecto es Instrumentino un *framework* de código abierto para desarrollar programas con GUI personalizados para controlar sistemas y experimentos, Instrumentino está escrito en Python, un lenguaje de programación de alto nivel que será descrito más adelante. El programa que permite al usuario controlar los diversos componentes de un sistema, al igual que permite crear secuencias de ejecución automáticas. Además de presentar los datos gráficamente también los guarda automáticamente en formato .csv.

Inicialmente, fue desarrollado para responder las necesidades de control de un sistema en un grupo de investigación, con el tiempo se convirtió en el programa de referencia para el control de sistemas en ese grupo y ya han realizado más de una docena de proyectos.

Instrumentino se describe en detalle en [1]. El código fuente está disponible en el servicio de alojamiento del repositorio de GitHub, y las adiciones contribuidas por el usuario pueden ser depositadas allí también.

4.1.2 Comunicación PC-Arduino

La comunicación entre el PC y el Arduino se realiza a través del puerto USB de ambos, mediante los programas Instrumentino y Controlino. Instrumentino envía al Controlino los comandos concretos, para algunos se espera una respuesta y para otros únicamente la confirmación de recibo.

El listado de comandos que Instrumentino puede enviar al Controlino es el siguiente:

Set, *Reset*, *BlinkPin*, *Read*, *Write*, *SetPwmFreq*, *PidRelayCreate*, *PidRelaySet*, *PidRelayTune*, *PidRelayEnable*, *HarfSerConnect*, *SoftSerConnect*, *SerSend*, *SerRecive* y *I2cWrite*.

Con el fin de que Arduino lea estos comandos, el *Sketch* (llamado Controlino) escucha constantemente los comandos textuales entrantes y actúa sobre ellos. Cada comando enviado del Instrumentino al Controlino es una cadena de Strings, terminado por un carácter de retorno de carro (CR).

Todos los comandos empiezan con una palabra que indica la función a realizar (*set*, *reset*, *write*, etc.), seguida de unos parámetros. Cuando se recibe un carácter CR, Controlino analiza la cadena recibida y actúa en función de ella. Al finalizar la ejecución de un comando, se responde con los datos pertinentes seguido de una cadena de caracteres, '' done! ''.

Por ejemplo si el usuario quiere leer el pin analogico A3, Instrumentino envia al Controlino 'Read A3', y este último lee el nivel de voltaje en el pin analógico A3 y

devuelve el resultado al Instrumentino. En otro caso en el que el usuario quiere modificar el pin digital D5 a nivel lógico 0, desde el Instrumentino se envía 'Write D5 0' al Controlino que establece el pin D5 a 0.

4.1.3 Archivo de configuración del sistema

El Archivo de configuración del sistema es un código escrito en Python que sirve para definir cómo es el sistema, qué componentes lo definen y a su vez que variables definen a cada componente. Cada una de estas variables se caracteriza por su rango de funcionamiento y sus unidades físicas (por ejemplo 0-100 °C), y las conexiones eléctricas necesarias para cada parámetro, un pin de entrada y un pin de salida. Esta información es todo lo que se necesita para traducir comandos de usuario de alto nivel a la operación de bajo nivel.

Los datos de configuración del sistema están compuestos de dos partes. La primera es una lista de componentes del sistema, acompañado de la información necesaria para su funcionamiento. Las clases apropiadas de Python describen los componentes y contienen la información esencial.

La segunda lista contiene acciones básicas significativas que el sistema debe realizar. Se trata de piezas cortas de código Python (funciones) que logran tareas básicas.

4.1.4 La interfaz gráfica de usuario

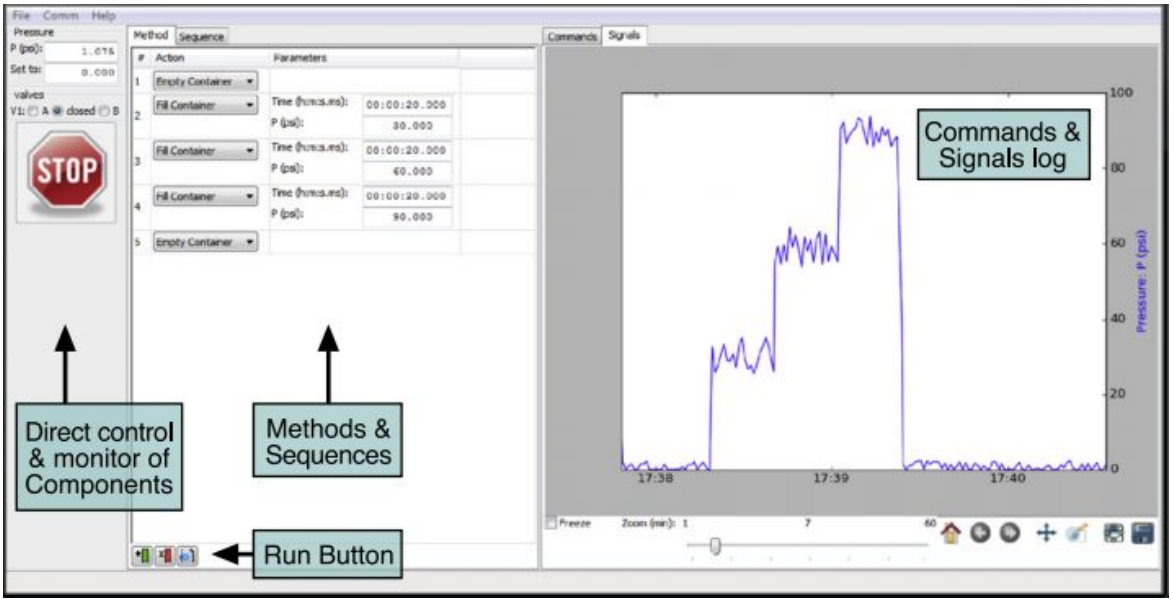

Figura 7: Interfaz gráfica de usuario [1]

La Fig. 7 muestra la interfaz gráfica de usuario que tiene siempre un formato uniforme.

La ventana principal se divide en tres subventanas: la de componente, la de automatización y la de registro.

4.1.4.1 La subventana de componentes

Proporciona la información de los componentes del sistema, se puede leer su valor y/o escribir el valor deseado.
Los distintos componentes se pueden juntar en grupos según se desee. También es posible que un único componente del sistema tenga varias variables.

4.1.4.2 La subventana de automatización

La siguiente es la subventana de automatización, en la que los métodos y secuencias se pueden definir y ejecutar. Un método es una lista extensible de las acciones definidas en el archivo de configuración del sistema. Una vez definido, un método se puede guardar como un archivo .csv para uso posterior.
Una Secuencia es simplemente una lista donde cada elemento de la lista se puede definir para que se repita varias veces, si es necesario, para construir largas ejecuciones operativas que pueden durar días.

4.1.4.3 La subventana de registro

Esta última subventana se utiliza para presentar las gráficas al usuario. Mientras el software se está ejecutando, los datos de las variables del sistema se trazan de manera superpuesta como series temporales.
A cada traza se le asigna un color y grosor de línea automáticamente. También se puede activar o desactivar su visibilidad haciendo clic en la entrada correspondiente en la leyenda del gráfico. Todos los trazos se dibujan en un eje vertical común, que refleja los valores actuales como un porcentaje del rango de variables relevantes. Los resultados positivos se representan como líneas continuas y los resultados negativos como líneas discontinuas.
Un conjunto de controles permite al usuario congelar la línea de tiempo y hacer zoom en diferentes áreas de interés en la gráfica. Las salidas de ambos registros se guardan automáticamente como archivos .csv y .txt respectivamente, utilizando la fecha y hora actuales como nombre de archivo.

4.2 Arduino IDE

El lenguaje de programación de Arduino es un *subset* de instrucciones C/C++. Para compilarlo, se dispone de un IDE (*Integrated Development Environment*) que se puede descargar desde la página oficial. Este programa incluye todas las herramientas de programación de todas las tarjetas oficiales, un editor de texto para el código, y diversas herramientas para poder depurar el código. Esta herramienta es compatible para Windows, Linux y Mac.

4.3 Python

Su implementación fue iniciada en diciembre de 1989 por Guido van Rossum, como un sucesor al lenguaje de programación ABC, capaz de manejar excepciones e interactuar con el sistema operativo Amoeba.
El lenguaje de programación Python es uno de los más usados en el mundo, es el tercero más popular en aquellos lenguajes que no basan su sintaxis gramatical en C.

Python es un lenguaje de programación interpretado cuya filosofía es hacer una sintaxis que favorezca un código legible. Se trata de un lenguaje de programación multiparadigma, permite varios estilos: programación orientada a objetos, programación imperativa y programación funcional.
También es utilizado en la computación científica gracias a librerías como NumPy, SciPy y Matplotlib.

4.4 Programación orientada a objetos

La programación orientada a objetos (POO) es una forma de programar que intenta ajustar las cosas a como son en la vida real.

Está basada en varias técnicas, incluyendo herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento.

Los objetos contienen toda la información que permite definirlos e identificarlos, para diferenciarse de otros objetos pertenecientes a otras clases o de la misma clase. Estos objetos a su vez disponen de métodos, mecanismos de interacción para comunicarse entre ellos.

A diferencia de la programación tradicional, en la que lo único que se busca es el procesamiento de unos datos de entrada para obtener unos datos de salida, en el caso de la POO, se definen objetos para que luego interactúen entre ellos.

5. Desarrollo

El desarrollo del proyecto se puede dividir en tres partes, en primer lugar la ingeniería inversa aplicada al Instrumentino, con el fin de entender cómo funciona y poder adaptarlo a nuestras necesidades modificando las partes deseadas.

Por otro lado, el siguiente paso que se ha realizado ha sido comprobar el correcto funcionamiento del Instrumentino con cada uno de los componentes conectados individualmente al Arduino.

Y finalmente se han integrado todos los dispositivos y se ha adecuado la presentación final de la interfaz gráfica a las necesidades del proyecto.

5.1 Ingeniería inversa

Instrumentino es un programa que cuenta con más de 30 archivos distintos, y más de 10.000 líneas de código además de usar varias librerías, por lo que los esfuerzos se han centrado en entender la parte del código útil para realizar cambios necesarios, con el fin de cumplir los objetivos del proyecto.

Con el fin de tener una visión más general se incluye la Fig. 8 donde podemos ver cómo están estructuradas las carpetas y ficheros más relevantes.

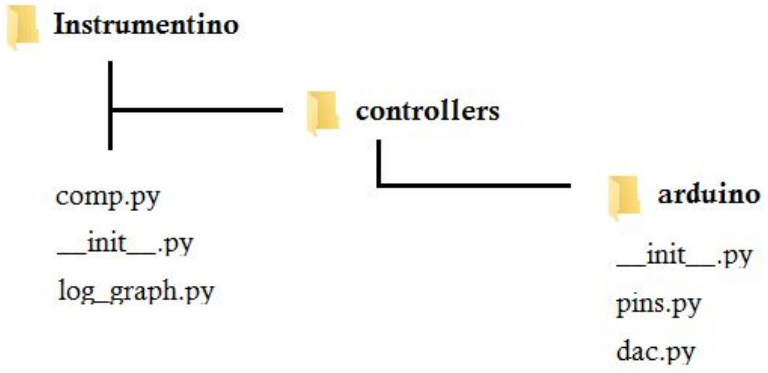

Figura 8: Organización de las carpetas y archivos más relevantes del Instrumentino

En la primera carpeta nombrada Instrumentino encontramos un conjunto de ficheros, en el primero de ellos *comp.py* se encuentra la clase *SysVar*, la clase original de cualquier variable que se quiera generar. En el mismo fichero también se encuentran las clases *SysVarAnalog* y *SysVarDigital* que como el mismo nombre indican depende de si la variable es del tipo analógico o digital será iniciada una o la otra. Por otro lado el fichero contiene la clase *SysComp,* un componente del sistema, que tiene variables, y se representa en la interfaz como un panel que contiene los paneles de las variables.

Seguimos por el fichero *log_graph.py*, este contiene una clase llamada *LogGraphPanel* la cual se encarga de generar la gráfica. y finalmente en esta misma carpeta el fichero *__init__.py* que inicia la aplicación mediante la clase *InstrumentinoApp.*

Dentro de la carpeta *Instrumentino* se encuentra también la carpeta *controllers* y dentro de esta está la carpeta *arduino.* Finalmente en esta encontramos un conjunto de archivos. El primero de ellos llamado también *__init__.py* este fichero contiene una clase llamada *Arduino* que implementa la interfaz entre el pc y el arduino. Por otro lado también contiene las clases como *SysVarAnalogArduino* que usan de referencia un objeto de la clase vista anteriormente *SysVarAnalog.* Y finalmente se encuentra la clase *SysCompArduino* que recibe como referencia la clase *SysComp* antes mencionada.

A continuación se encuentra el fichero *pins.py* con dos clases *DigitalPins* y *AnalogPins* ambas inician *SysCompArduino*, pero enviando distintas referencias.

5.1.1 Iniciación

En primer lugar se ejecuta el archivo de configuración. El código se inicia con el *main* principal que crea un objeto de tipo *System*. Al iniciarse este objeto, crea los objetos comps y actions que representan los componentes y las acciones respectivamente.

```
if __name__ == '__main__':
    # run the program
    System()

class System(Instrument):
    def __init__(self):
        comps = (analPins,)
        actions = (SysActionSetPins(),)
        name = 'Arduino simple example'
        description = 'A simple example using only an Arduino. Analog pins 1,2 s
        version = '1.0'

        Instrument.__init__(self, comps, actions, version, name, description)
```

La clase *System* inicializa automáticamente un objeto de tipo *Instrument*, al cual se le envían los objetos *comps* y *actions*, definidos en el archivo de configuración. El objeto *Instrument* se encuentra definido dentro de carpeta instrumentino en el archivo llamado *__init__.py*.

```
class Instrument():
    '''
    an instrument parent class
    '''
    def __init__(self, comps, actions, version='1.0', name='Instrume
        self.comps = comps
        self.actions = actions
        self.version = version
        self.name = name
        self.description = description

        self.StartApp()

    def GetSystemUid(self):
        '''
        Return a unique id for the instrument
        '''
        return self.name + self.version + self.description

    def StartApp(self):
        '''
        Run application
        '''
        app = InstrumentinoApp(self)
        app.MainLoop()
```

La clase *Instrument* contiene el método *StartApp* el cual se inicia automáticamente al crear un objeto de tipo *Instrument*. Este método crea un objeto de tipo *InstrumentinoApp* al que llama *app* y después pasa a recorrer el *MainLoop* donde se encarga de inicializar y actualizar la interfaz gráfica.

```
class InstrumentinoApp(wx.App):
    '''
    This class implements the application
    '''
    monitorUpdateDelayMilisec = Arduino.cacheReadDelayMilisec
    updateFrequency = 1000 / monitorUpdateDelayMilisec

    def __init__(self, system):
        self.system = system
        self.sysComps = self.system.comps
        self.sysActions = self.system.actions
        wx.App.__init__(self, False)
```

La clase *InstrumentinoApp* inicializa *wx*, la librería *wxPython* que se encarga de la interfaz gráfica.

5.1.2 Componentes y variables

Por otro lado tenemos el objeto *comp*, que como se ha visto anteriormente se crea en el archivo de configuración, *comp* puede estar formado por diversos tipos de componentes, *analPins*, *digiPins*, *spi_dac_channels* o *i2c_dac_channels*.
En cuanto a la escritura y lectura de pins se ha de diferenciar el funcionamiento dependiendo de si los pines son analógicos o digitales. Dentro de la carpeta instrumentino/controllers/arduino se encuentra el archivo llamado *pins.py* ahí se encuentran los objetos *DigitalPins* y *AnalogPins* dependiendo de si los pines son digitales o analógicos.

```
class DigitalPins(SysCompArduino):
    def __init__(self, name, digiVars=()):
        SysCompArduino.__init__(self, name, digiVars, "turn on/off Arduino digital pins")


class AnalogPins(SysCompArduino):
    def __init__(self, name, analogVars=()):
        SysCompArduino.__init__(self, name, analogVars, "set Arduino analog pins")
```

De estas clases nos interesan los parámetros *digiVars* y *analogVars* para los cuales hay una subclase donde se introducen todos los parámetros de los pines, se pondrá un ejemplo:

```
analPins = AnalogPins('analog pins',
                      (SysVarAnalogArduinoUnipolar('unipolar +',[0,5],pinAnal_unipolarPositive,None, units='V'),
                       SysVarAnalogArduinoUnipolar('unipolar -',[-5,0],pinAnal_unipolarNegative,None, units='V'),
```

En el ejemplo, se envia a *AnalogPin*s *SysVarAnalogArduinoUnipolar* una clase que se encuentra en la misma carpeta arduino pero en el archivo *__init__.py*. Se deben crear nuevas clases para los nuevos sistemas que se desarrollan a no ser que las pocas clases ya existentes sean útiles. En el caso del ejemplo el pin analógico tiene un rango de 0 a 5 y solamente dispone de una entrada llamada "pinAnal_unipolarPositive".

SysVarAnalogArduinoUnipolar es una subclase de *SysVarAnalogArduino* que a su vez es una subclase de *SysVarAnalog,* esta última ubicada en la carpeta instrumentino y en el archivo *comp.py*. También subclase de *SysVar* el objeto original.

Cada una de las subclases añaden funcionalidades al objeto original, por ejemplo partiendo del objeto SysVar se pueden crear los objetos *SysVarAnalog* y *SysVarDigital* y así sucesivamente creando finalmente objetos como *SysVarAnalogArduinoUnipolar* personalizados para cada variable necesaria en el proyecto.

En el caso de las variables es necesario que las subclases de SysVar tengan los metodos *Update()*, *GetFunc()*, *SetFunc(value)* y *OnEdit(event)*. Los más importantes *GetFunc* y *SetFunc* para leer o escribir en la variable.

5.1.3 *GetFunc* y el *SetFunc*

En el caso de los pines digitales las subclases definen el *SetFunc* y el *GetFunc* de la siguiente manera:

```
def GetFunc(self):
    if self.pin != None:
        value = self.GetController().DigitalRead(self.pin)
        return self.valueToState[value] if value != None else None
    else:
        return self.lastSetState

def SetFunc(self, state):
    self.lastSetState = state
    if self.pin != None:
        self.GetController().DigitalWrite(self.pin, self.stateToValue[state])
```

Mediante el método *DigitalWrite* se lee y escribe el valor del pin deseado.

```
def DigitalWrite(self, pin, value):
    '''
    Set the voltage level of a digital output pin to value (1 to HIGH and 0 to low)
    '''
    self._sendData('Write %d digi %d'%(pin, value), wait=True)
```

Este método envía la información del pin y del valor y añade la información que el Controlino necesita para realizar la función oportuna *Write digi*. Como vemos en el código esta función a la vez llama la función *sendData*, que básicamente envía la información por el puerto serie, en caso de esperar una respuesta la guarda en una variable llamada *rxData* y después espera la cadena de Strings “done!” para verificar la transmisión con el Controlino.

Para el caso de la comunicación analógica el *SetFunc* y el *GetFunc* son algo diferentes.

```
def GetFunc(self):
    fraction = self.GetController().AnalogReadFraction(self.pinIn, self.pinInVoltsMax, self.pinInVoltsMin)
    sign = 1 if self.GetPolarityPositiveFunc() else -1
    return sign * (self.GetUnipolarMin() + (self.GetUnipolarRange() * fraction)) if fraction != None else None

def SetFunc(self, value):
    fraction = (abs(value) - self.GetUnipolarMin()) / self.GetUnipolarRange()
    if self.pinOut != None:
        self.GetController().AnalogWriteFraction(self.pinOut, fraction, self.pinOutVoltsMax, self.pinOutVoltsMin)
    elif self.I2cDac != None:
        minV = self.pinOutVoltsMin
        maxV = self.pinOutVoltsMax
        self.I2cDac.WriteFraction((minV + (maxV - minV) * fraction) / 5, self.GetController())
```

En primer lugar se usan las funciones *AnalogReadFraction* y *AnalogWriteFraction* las cuales también llaman a otras funciones como *AnalogReadVolts* y esta finalmente a *AnalogRead*. En el caso del *AnalogWriteFraction*, después de llamar a algunas funciones se usa finalmente la función sendData.

```
def AnalogRead(self, pin):
    '''
    Read the voltage level of an analog input pin using a 10-bit value

    Returns: value between 0-1023
    '''
    # Get value from cache if possible, and if not, add it to the wish-list
    try:
        return self.pinValuesCache['A' + str(pin)]
    except KeyError:
        self.pinValuesCache['A' + str(pin)] = 0
```

Donde *pinValuesCache* se va actualizando con las medidas analógicas del pin deseado.

```
def CacheUpdate(self, event):
    pins = ''
    # save the keys in case they change while we read
    keys = self.pinValuesCache.keys()
    for k in keys:
        pins += k + ' '

    valuesStr = self._sendData('Read %s'%(pins.strip()))
    if valuesStr == None:
        return

    values = valuesStr.split(' ')

    # values are received in the same order we asker for them
    try:
        for key, val in zip(keys, values):
            self.pinValuesCache[key] = int(val)
    except ValueError:
        print 'Read %s'%(pins.strip())
        print 'received: ' + valuesStr
```

Y para la lectura en caso de usar comunicación SPI o I2C a bajo nivel se usa también la función *sendData*, pero a más alto nivel se incluye información *I2cDac* a la variable.

```
i2c_dac_channels = AnalogPins('I2C DAC channels',
                              (SysVarAnalogArduinoUnipolar('0x2C',[0,5],i2c_dac_anal_in,None, units='V', I2cDac=i2c_dac),))
```

```
i2c_dac = DacI2cMAX517(0x2C)
i2c_dac_anal_in = 13
```

Mediante la creación de un objeto, *DacI2cMAX517* el cual tiene definido el método de lectura I2C llamado *WriteFraction*

```
class DacI2cMAX517(ArduinoDac):
    '''An I2C DAC connected to an Arduino.
    '''
    def __init__(self, address):
        self.address = address
        self.dacBits = 8
        super(DacI2cMAX517, self).__init__(self.dacBits)

    def WriteFraction(self, fraction, controller):
        controller.I2cWrite(self.address, (0, self.maxVal * fraction,))
```

5.2 Desarrollo segmentado por componentes

El siguiente paso que se ha realizado ha sido comprobar el correcto funcionamiento del Instrumentino con cada uno de los componentes conectados individualmente al Arduino. Primero se comprueba su funcionalidad a través del Monitor serie del programa IDE y a continuación se añade también el Instrumentino.
Empezamos por comprobar el funcionamiento del Instrumentino con el Arduino Due, para esto se utilizó el ejemplo que viene con el Instrumentino.

El único cambio realizado hasta el momento ha sido cambiar en el Controlino el nombre de la placa Arduino que se utiliza, ya que en vez de usar el Arduino Uno se usa el Arduino Due. Al contrario de lo esperado, no funcionó.

En el caso de la comunicación serial en el Arduino Due tiene implementados UART y USART de HardwareSerial, se realizaron los siguientes cambios en el *Sketch* de Controlino:

```
extern HardwareSerial Serial1  → extern USARTClass Serial1
extern HardwareSerial Serial2  → extern USARTClass Serial2
extern HardwareSerial Serial3  → extern USARTClass Serial3
extern HardwareSerial Serial   → extern UARTClass Serial1
```

5.2.1 Potenciómetro

Una vez solventado el problema con la comunicación entre el Arduino Due e Instrumentino el ejemplo ya funcionaba correctamente, se realizaron las comprobaciones con el potenciómetro, ya que la escritura a través de la comunicación SPI ya estaba implementada en el Instrumentino.

El primer paso fue comprobar el funcionamiento del potenciómetro conectándolo a través de la comunicación SPI únicamente al Arduino, el Sketch realizado se puede ver en el (Anexo A.6) donde la comunicación está implementada usando el pin 53 como CS. Modificando el valor de la variable *valor,* se modifica el valor del potenciómetro siendo para *valor*=0 de 0 *ohms* y para *valor*=250 de 10 k*ohms*.

Seguidamente, para la comunicación con el Instrumentino se realizaron los siguientes cambios. En primer lugar como se ha comentado con anterioridad al usar el arduino Due el código para transmitir mediante SPI es distinto, por lo que se tuvo que realizar los siguiente cambio de instrucciones, la instrucción *SPI.transfer(strtol(argV[i], NULL, 10))* utilizada se reemplazó por *SPI.transfer(53, strtol(argV[i], NULL, 10))* al igual que la instrucción que inicia la comunicación SPI, *SPI.begin()*, que en el caso del Arduino due se tiene que iniciar con 4,10 o 53 dependiendo del canal SPI que se use, en este caso SPI.begin(53).

En esta parte surge el conflicto de o bien iniciar y terminar constantemente la comunicación SPI o simplemente modificar la parte del Controlino para que se inicie el SPI deseado que es por la que me decanté.

Finalmente siguiendo el ejemplo de la clase *DacSpiMCP4922*, se modificó, para crear la clase del componente Potenciómetro. (Anexo B.2)

5.2.2 Sensor de Temperatura

En primer lugar se realizó el Sketch en Arduino para comunicarse con el sensor (anexo A.5), verificando su correcto funcionamiento con el monitor serie. Para el correcto funcionamiento requiere de la recepción de dos bytes por lo que se ha de incluir *SPI_CONTINUE* en la transmisión SPI.

```
byte response1 = SPI.transfer(53, 0x00, SPI_CONTINUE);
byte response2 = SPI.transfer(53, 0x00);
```

A continuación se juntan los dos bytes en un mismo integer y se prescinde de los dos bits más significativos, finalmente se multiplica por 0,031125(LSB) para obtener el resultado en grados Celsius.

```
value=response1<<8;
value = value | response2;
value = value>>2;
value=value*0.031125;
```

Después se pasó a realizar la medida del sensor de temperatura mediante Instrumentino, el envío de datos a través de la comunicación ya estaba implementado, pero faltaba por implementar la lectura de datos. Este fue un punto importante del proyecto, después de la ingeniería inversa aplicada al código de Instrumentino la teoría para realizar este punto era clara.

Por un lado crear en el Controlino una función llamada *cmdSpiRead* igual que la usada con el potenciómetro pero que retornara un valor al instrumentino *return SPI.transfer(53, strtol(argV[i], NULL, 10))*. Esta función al igual que las demás funciones del Controlino se ejecutaría al recibir de Instrumentino el String deseado, por ejemplo *SpiRead*.

Mientras que por la parte del Instrumentino se requiere seguir unos pasos similares a los realizados para la escritura SPI, crear una función *SpiRead* en la clase Arduino igual que la ya existente *SpiWrite* pero guardando la información recibida del Controlino y devolviendola a quien llamara la función con un return.

Siguiendo también los pasos del *SpiRead* pero realizando los cambios que actualmente contiene el "*Set*" en el "Get".

Después de realizar todos los cambios oportunos y probar variantes seguía sin funcionar, y como el proyecto debía avanzar se encontró una solución funcional.

Se decidió utilizar el método de transmisión que se usa para los pines digitales que era fácil y entendible, la función lee la información transmitida por el Instrumentino, si el pin es analogico o digital y el número de pin. A continuación el Controlino hace una lectura del voltaje en el número de pin deseado y devuelve el valor por el puerto serie.

El número de pin es fijado en el archivo de configuración del Instrumentino por lo que es fácilmente variable, entonces se envía un número de pin que esté en desuso o sea inexistente en la placa y se añade en la función *cmdRead* un código en el que si recibimos el número de pin antes mencionado realice la comunicación SPI y nos devuelva el resultado.

El código también acondiciona los datos antes de enviarlos para que una vez sean recibidos por el Instrumentino la conversión de binario a integer sea sencilla.

A continuación se creó una nueva clase en el Instrumentino, *SysVarAnalogArduinoUnipolarTemp,* donde el método *GetFunc* recibe la información del sensor en binario y esta información es convertida a Integer multiplicando por la sensiblidad de 0,031125ºC/LSB, en medida de lo posible a lo largo de todo el proyecto se realizan todos los cálculos en el Instrumentino, ya que el ordenador dispone de una potencia mucho mayor a la del Arduino.

```
def GetFunc(self):
    fraction = self.GetController().AnalogReadFraction(self.pinIn, self.pinInVoltsMax
    return fraction*0.031125 if fraction != None else None
```

Para obtener el valor en las unidades correctas, también se tuvieron que variar las

funciones *AnalogReadFraction* y *AnalogReadVolts* porque estas están configuradas para leer el voltaje analogico, sin embargo como en este proyecto no necesitamos la lectura de pines análogicos se modificaron directamente estas funciones, en caso de que hubiéramos necesitado estas funciones se hubieran creado dos de nuevas para la lectura del sensor. Podemos ver la modificación a continuación.

```
def AnalogReadVolts(self, pin):
    '''
    Read the voltage level of an analog input pin directly
    returns - value between 0-5 (volts)
    '''
    value = self.AnalogRead(pin)
    if value != None:
        #return self.PIN_VOLT_MAX * value / self.ANAL_IN_VAL_MAX
         return value

def AnalogReadFraction(self, pin, maxV=5, minV=0):
    '''
    Read the voltage level of an analog input pin using a fraction
    minV - minimal voltage
    maxV - maximal voltage
    returns - value between 0-1
    '''
    value = self.AnalogReadVolts(pin)
    if value != None:
        #return (value - minV) / (maxV - minV)
         return value
```

donde el código que aparece comentado, es prescindible para nuestras necesidades.

Después de estos cambios las medidas del sensor de temperatura la comunicación con el Instrumentino se realizaban de forma correcta.

5.2.3 Acelerómetro

A continuación con el acelerómetro el procedimiento es similar al anterior pero utilizando la comunicación I2C. En primer lugar se comprobó el funcionamiento del acelerómetro únicamente con el Arduino, para ello se utilizó la librería LIS331 como se muestra en el código del Anexo A.3.

Para la comunicación con el Instrumentino se siguieron los pasos del apartado anterior usando una clase igual a la *SysVarAnalogArduinoUnipolarTemp*, pero sin realizar la multiplicación por el valor del *LSB*.

5.2.4 FPGA y regulador LDO

Y por último la FPGA y el LDO mediante la segunda comunicación I2C. Siguiendo los mismos pasos que para los dispositivos anteriores primero se conectarán únicamente

con el arduino para verificar su funcionamiento y después se incluirá el Instrumentino.

Programando el arduino con el código que se puede encontrar en el Anexo A.1 y mediante un tester confirmamos el correcto funcionamiento del LDO, el código se comunica con la dirección 0x60, dirección del regulador LDO y mediante direcciones de los registros 0x39 y 0x3A configura los voltajes de 3,3V y 2,5V.

A continuación mediante el código del Anexo A.4 el arduino se comunica con la FPGA, la dirección de la FPGA es la 0x30 y se mira el valor guardado en el registro 0x81 (*Code version*), este registro tiene asignado el valor 0x21, se le pregunta a la FPGA por el valor de este registro y al recibir el valor 0x21 se verifica el correcto funcionamiento de la comunicación.
El primer I2C del Arduino Due se inicia mediante *Wire.begin()*, por otro lado para iniciar el segundo se requiere de la instrucción *Wire1.begin()*.

En el caso del Arduino Due al disponer de dos comunicaciones I2C para referirse a la primera es mediante el librería Wire y para la segunda Wire1, estas dos librerías permiten al Arduino comunicarse con dispositivos mediante I2C, otra diferencia es que la primera comunicación I2C la misma placa del Arduino Due contiene las resistencias de pull up de 1,5 k*ohms* necesarias, sin embargo la segunda comunicación I2C requiere que se incorporen en el circuito.

5.3 Integración

5.3.1 Integración

Para el último paso del desarrollo del proyecto se requiere utilizar la placa base con todos los dispositivos conectados y realizar el archivo de configuración del sistema haciendo las modificaciones correctas para obtener los resultados en la interfaz gráfica de la forma deseada.

Se empezó por cambiar el código del acelerómetro por el del magnetómetro en el Controlino, ambos muy parecidos.

A continuación se comprobó el correcto funcionamiento de la interfaz gráfica con todos los dispositivos conectados al mismo tiempo, al existir un conflicto en la placa base con las comunicaciones I2C no funcionó. Un problema de diseño de la placa hace que cuando la FPGA está programada y funcionando, las líneas I2C de los sensores magnéticos de referencia (comerciales) reciben interferencias suficientes como para provocar una falsa detección de intento de lectura, lo que hace que algún dispositivo deje la línea a '0' lógico y afectando también a la línea I2C secundaria del Arduino Due.
Como este problema está más allá del tema de éste trabajo lo que se ha hecho ha sido comprobar el correcto funcionamiento sin usar las dos líneas I2c simultáneamente.

Fueron añadidas al Controlino las variables *ScaleLIS3mDL* y *ValPot*, destinadas a inicializar el magnetómetro y el potenciómetro con los valores deseados de forma práctica.

```
#define ScaleLIS3MDL 0x00
```

```
#define ValPot 0xFA
```

Posteriormente estas variables son llamadas en el *setup()* del programa por las funciones *mag.writeReg(**LIS3MDL**::CTRL_REG2,ScaleLIS3MDL);* y *digitalWrite(8, LOW); SPI.transfer(53, ValPot); digitalWrite(8, HIGH);*

Por otro lado, el pin 13 que se encarga de activar la lectura del sensor de temperatura en algunos casos daba problemas, se encontró que el Instrumentino usaba el pin 13 en la función *cmdBlinkPin* del Controlino para hacer parpadear un led, al eliminar esta parte, las lecturas se realizan de forma correcta.

El archivo de configuración con el que se integran todos los componentes en una misma interfaz gráfica se puede ver en el Anexo B.3.

5.3.2 Ajuste de la gráfica

El eje Y de la gràfica que genera Instrumentino por defecto está definida en tanto por ciento los valores positivos en una traza continua y los negativos en una traza discontinua como se ha comentado en la descripción del Instrumentino. Para el proyecto en cuestión en que los ejes del magnetómetro muestran tanto valores positivos como negativos el gráfico resulta poco claro, además de que al estar midiendo campo magnético, nos interesa que las unidades estén en µT.

Para eso se modificó en el archivo *log_graph.py*, *self.axes.set_ybound(0,100)* por los valores del eje Y deseados, el máximo rango de valores que se quieren mostrar vienen definidos por el sensor magnético operando con una escala de ±16 G, por lo que el valor máximo y mínimo es $\pm 1600 \mu T$ por lo que será ajustada a $\pm 2000 \mu T$, *self.axes.set_ybound(-2000,2000)*.

El sensor magnético cuenta con 15 bits para transmitir los datos, 15 bits → $2^{15} = 32768$

$$32768\ LSB * \frac{1\ G}{1711\ LSB} * \frac{1000\ mG}{1G} * \frac{0{,}1\ \mu T}{1\ mG} = 1915{,}1373\ \mu T \qquad (1)$$

Como se observa en (1) en realidad el valor máximo será 1915,1373µT

Por otro lado en el mismo archivo también se modificó el método *NormalizePositiveValue* para que no devolviera los valores en tanto por ciento como se muestra a continuación.

Este fragmento de código es el original

```
def NormalizePositiveValue(self, value, yRange):
    # unipolar range [X, Y] or [-X, -Y]
    relevantEdge = yRange[0] if yRange[0] >= 0 else yRange[1]
    return abs(value - relevantEdge) / abs(yRange[1] - yRange[0]) * 100
```

Y a continuación tenemos el método después de modificarse

```
def NormalizePositiveValue(self, value, yRange):
    return value
```

5.3.3 Ajuste de la sensibilidad del magnetómetro

Por otro lado, el pin 13 que se encarga de activar la lectura del sensor de temperatura en algún caso daba problemas, se encontró que el Instrumentino usaba el pin 13 en una función para hacer parpadear un led, al eliminar esta parte, las lecturas ya son todas correctas.

El magnetómetro LIS3MDL como se ha descrito anteriormente tiene un rango y una sensibilidad ajustables, uno de los objetivos del proyecto era poder variar estos valores a través de la interfaz gráfica, para no tener que reiniciar las medidas cada vez que se realizará un cambio.

Para llevar a cabo este punto se implementaron las clases oportunas en el Instrumentino que dependiendo del valor de una variable aumenta el rango de valores y por tanto disminuye el peso del LSB.

Se añadió a la función “cmdWrite” del Controlino las líneas de código oportunas para que dependiendo del valor que recibiera modifique la configuración del sensor magnético, a continuación se puede observar el código.

```
void cmdWrite(char **argV) {
        int pin = strtol(argV[1], NULL, 10);
        char* type = argV[2];
        int value = strtol(argV[3], NULL, 10);
        if (value==4){
                mag.writeReg(LIS3MDL::CTRL_REG2,0x00);
        }else if (value==8){
                mag.writeReg(LIS3MDL::CTRL_REG2,0x20);
        }else if (value==12){
                 mag.writeReg(LIS3MDL::CTRL_REG2,0x40);
        }else if (value==16){
                mag.writeReg(LIS3MDL::CTRL_REG2,0x60);
        }else if (strcasecmp(type, "digi") == 0) {
                if (value == 0) {
                        digitalWrite(pin, LOW);
                } else {
                        digitalWrite(pin, HIGH);
                }
        } else if (strcasecmp(type, "anal") == 0) {
                analogWrite(pin, max(0, min(ANAL_OUT_VAL_MAX, value)));
         } else {
        return;  }
}
```

Por otra parte también se requirió modificar el código del Instrumentino, en concreto la clase *SysVarAnaloArduinoMag4,* para que el valor resultante esté en las unidades correctas. A continuación se puede ver el código modificado, donde la variable *sen_mag* es usada para obtener el resultado en las unidades deseadas.

```
def GetFunc(self):
    fraction = self.GetController().AnalogReadFraction(self.pinIn, self.pinInVoltsMax, self.pinInVoltsMin)
    sign = 1 if self.GetPolarityPositiveFunc() else -1
    return sign * ((self.GetUnipolarMin() + (self.GetUnipolarRange() * fraction))/self.sen_mag)*100 if fraction

def SetFunc(self, value):
    fraction = (abs(value) - self.GetUnipolarMin()) / self.GetUnipolarRange()
    if value == 12:
        self.sen_mag =2281
    if value ==16:
        self.sen_mag =1711
    if value ==8:
        self.sen_mag =3421
    if value ==4:
        self.sen_mag =6842
    if self.pinOut != None:
        self.GetController().AnalogWriteFraction(self.pinOut, fraction, self.pinOutVoltsMax, self.pinOutVoltsMir
    elif self.I2cDac != None:
        minV = self.pinOutVoltsMin
        maxV = self.pinOutVoltsMax
        self.I2cDac.WriteFraction((minV + (maxV - minV) * fraction) / 5, self.GetController())
```

6 Resultados

6.1 Resultados

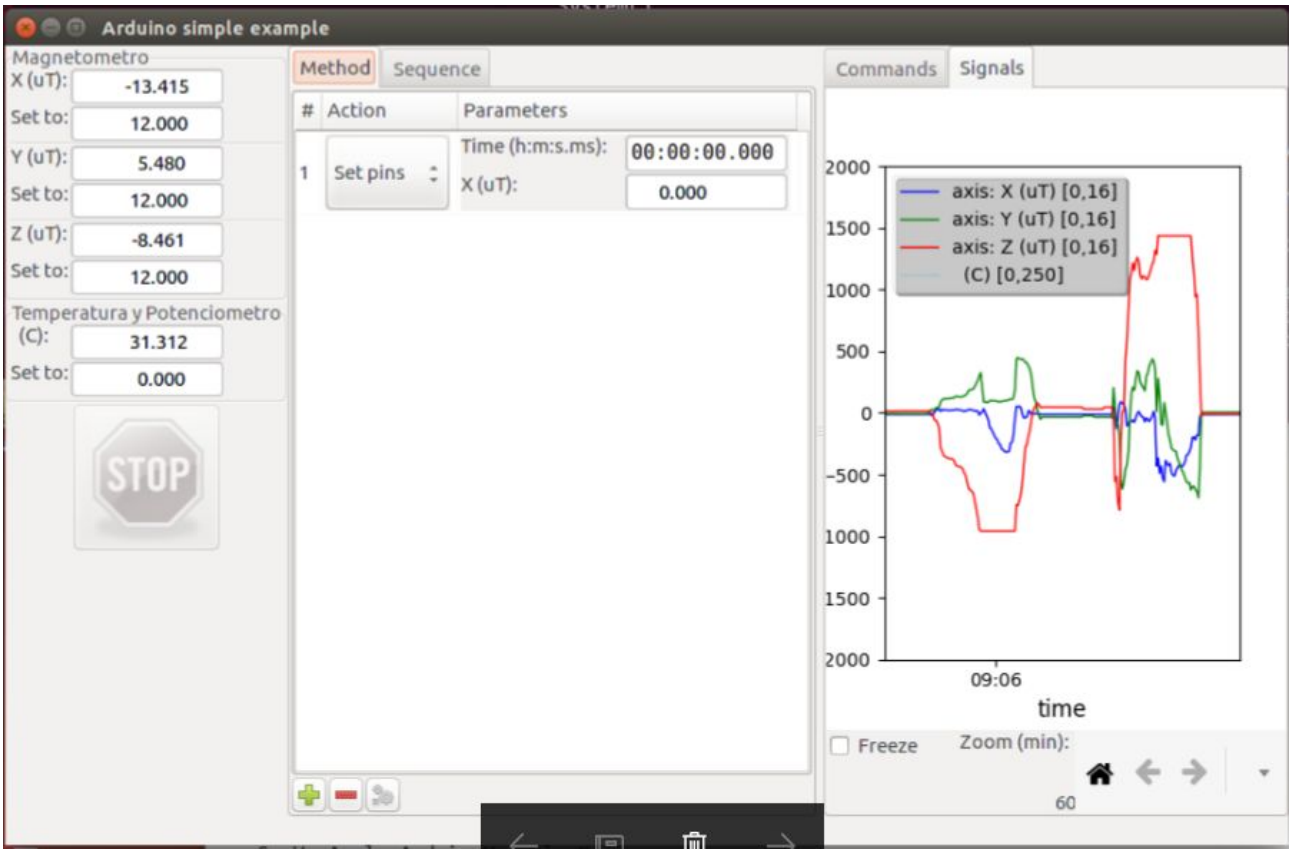

Figura 9: Interfaz gráfica resultante del proyecto

La Fig. 9 muestra la GUI resultante del proyecto, si nos fijamos en la parte de la izquierda se puede observar que se ha creado un componente llamado *Magnetometro* que tiene 3 variables nombradas *X*,*Y* y *Z* en las cuales se muestra el valor del campo magnético de cada uno de los ejes. También podemos modificar el rango de escalera del magnetómetro con la variable debajo de ellas escribiendo el valor deseado (4,8,12 o 16). Los valores de los 3 ejes en microTeslas y aunque en la interfaz gráfica solo aparecen 3 decimales el archivo csv. donde guardamos los datos se guardan los valores con todos los decimales (Fig. 10)

El siguiente componente es el llamado *Temperatura y Potenciómetro* el cual contiene dos variables, la primera de ellas respecto a la temperatura donde se visualiza el valor de la temperatura en ºC y por último la variable para variar el valor del potenciómetro introduciendo un valor deseado entre 0 y 250, equivalente a 0 *ohms* y 10 k*ohms* de forma lineal.

	A	B	C	D	E
1	time	axis: Y (T)	axis: X (T)	axis: Z (T)	Temperature: T (C)
2	11:30:42.601005	0.0876936568	-9.7632271266	-9.7486115171	30.346875
3	11:30:42.849847	0.2338497515	-9.529377375	-9.3686056709	30.346875
4	11:30:43.099705	-0.1461560947	-9.9532300497	-9.7924583455	30.346875
5	11:30:43.352766	-0.0584624379	-9.7193802982	-9.4562993277	30.31575

Figura 10: Ejemplo de un fichero guardado en formato csv.

Siguiendo en la parte de la derecha de Fig. 9 se encuentra la gráfica, con un eje Y comprendido entre -2000µT y 2000µT. En la primera parte de la traza de las líneas de la gráfica se observa como la gráfica se satura por entorno a los -800µT debido a que la escalera del magnetómetro estaba ajustada a $\pm 8G$, después se modificó el ajuste de la escalera a $\pm 12G$ y por esto se observa que se satura en torno a 1400µT

6.2 Reporte de las modificaciones a Github

Para finalizar con el proyecto se ha decidido colgar en github los cambios realizados en el Controlino para su correcto funcionamiento con la placa de Arduino Due, con el fin de que a partir de ahora otros usuarios no tengan el mismo contratiempo. En el caso de las comunicaciones implementadas en el Instrumentino, al ser modificaciones demasiado específicas para este proyecto se ha decidido no incluirlas.

Los cambios reportados han sido los siguientes, los cambios para el correcto funcionamiento de la comunicación Serial:

extern HardwareSerial Serial;

extern HardwareSerial Serial1;
extern HardwareSerial Serial2;
extern HardwareSerial Serial3;

por

extern UARTClass Serial;

extern USARTClass Serial1;
extern USARTClass Serial2;
extern USARTClass Serial3;

y la modificación en la lectura en caso de que se use la comunicación SPI cambiando:

SPI.transfer(strtol(argV[i], NULL, 10)); por *SPI.transfer(cs_pin, strtol(argV[i], NULL, 10));*

7. Conclusiones

A lo largo del proyecto he observado más de cerca el funcionamiento de los grupos de investigación y cómo se enfrentan a los problemas que se les presentan de forma ingeniosa, optimizando los recursos que tienen pero sin perder calidad en los análisis.

También he visto la frecuente necesidad de realizar pruebas, procesar, visualizar y almacenar los datos para posteriormente analizarlos y la necesidad por mejorar en este aspecto.

Instrumentino presenta unas buenas características para ser esa forma ingeniosa de resolver la necesidad, de una forma práctica y económica.

Sin embargo Instrumentino aún está muy lejos de programas como LabView, sí que presenta un solución práctica para actuar con sistemas simples, pero para aprender a usarlo se requiere un tiempo y compaginarlo con sistemas más complejos se requiere un aprendizaje.

Por otro lado si además se requiere modificar el código aún se hace más denso el trabajo, sobre todo si no se es un experto en Python y en la programación orientada a objetos.

Para concluir considero que con el tiempo dedicado y el trabajo aportado se hubiera podido realizar una interfaz gráfica que cumpliera con los objetivos propuestos, pero hacerla flexible para ser usada para otros proyectos hubiese dificultado su diseño. Instrumentino requiere una larga fase de adaptación, pero una vez acabada considero que es una buena herramienta para los futuros proyectos del grupo de investigación.

Bibliografía:

[1] I.J. Koenda, J. Sáiz, P.C. Hauser, “Instrumentino: An open-source modular Python framework for controlling Arduino based experimental instruments,” Computer Physics Communications, vol. 185, no. 10, pp. 2724-2729, June 2014.

[2] I.J Koenka, J. Sáiz, P.C. Hauser, “*Instrumentino*: An Open-Source Software for Scientific Instruments,”SCS Fall Meeting 2014, vol. 67, no. 4, pp. 172-175, 2014.

[3] pinodisco, Reading RPM From Arduino-based IR Tachometer With MATLAB GUI, http://www.instructables.com/id/Reading-RPM-from-Arduino-based-IR-Tachometer-with-/, February 2017

[4] M. Pérez, Arduino y LabVIEW, https://geekytheory.com/arduino-y-labview, February 2017

[5] techbitar, Using Visual Basic 2010 to Control Arduino Uno, http://www.instructables.com/id/Using-Visual-Basic-to-control-Arduino-Uno/, Februaary 2017.

[6] madshobye, Guino:Dashboard for Your Arduino, http://www.instructables.com/id/Guino-Dashboard-for-your-Arduino/, February 2017.

[7] Arduino Due, https://www.reichelt.com/de/en/Single-board-microcontroller/ARDUINO-DUE/3/index.html?ACTION=3&GROUPID=6667&ARTICLE=130169, April 2017

[8] *LIS3MDL magnetometer datasheet*, ST Microelectronics.

[9] *LIS95071 temperature sensor datasheet*, Texas Instruments.

[10] *LIS331DLH acelerometer datasheet*, ST Microelectronics.

[11] *AD5169 digital potentiometer*, Analog Devices.

[12] Serial Pheripheral Interface bus https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus, March 2017.

[13] Serial Pheripherial Interface https://es.wikipedia.org/wiki/Serial_Peripheral_Interface, March 2017.

[14] I2C https://es.wikipedia.org/wiki/I%C2%B2C, March 2017.

ANEXO A:

Código Arduino

1. Función para programar el el regulador de voltaje LDO:

```
void LDOs_I2C_config(void)
{
   Wire1.beginTransmission(0x60); // Comunicación con el dispositivo con la dirección 0x60

  Wire1.write(0x39); // Address 0x39
  Wire1.write(B10111); // Data 0x17 3V3
  int VDIG_log = Wire1.endTransmission(true);


  Wire1.beginTransmission(0x60);
  Wire1.write(0x3A); // Address 0x3A
  Wire1.write(B01111); // Data 0x0F 2V5
  int VANA_log = Wire1.endTransmission(true); // Ends transmission and releases bus
}
```

2. Comunicación entre el Arduino y el sensor magnético LIS3MDL usando la librería LIS3MDL.

```
#include "Wire.h"
#include <LIS3MDL.h>
LIS3MDL mag;

char report[80];
int16_t a, b ,c;
void setup() {
  Serial.begin(115200);
  Wire1.begin();
  LDOs_I2C_config;  //Configura el LDO

    if (!mag.init())
  {
    Serial.println("Failed to detect and initialize magnetometer!");
    while (1);
```

```
  }

  mag.enableDefault();
  mag.writeReg(LIS3MDL::CTRL_REG2,0x00); // Configura el rango y la sensibilidad
}

void loop() {
mag.read();
a=mag.m.x;
b=mag.m.y;
c=mag.m.z;
  snprintf(report, sizeof(report), "M: %6d %6d %6d",a, b, c);
  Serial.println(report);

}
```

3. Lectura de los datos del acelerómetro por parte del Arduino usando la librería LIS331.

```
#include <LIS331.h>
#include <Wire.h>
LIS331 lis;
void setup(){
    Wire.begin();
    lis.setPowerStatus(LR_POWER_NORM);
    lis.setXEnable(true);
    lis.setYEnable(true);
    lis.setZEnable(true);
    Serial.begin(115200);
}
void loop(){
int16_t x,y,z;
  digitalWrite(7, HIGH);
        lis.getXValue(&x);
        lis.getYValue(&y);
        lis.getZValue(&z);
        Serial.println("X Value: " x " milli Gs");
        Serial.println("Y Value: " y " milli Gs");
        Serial.println("Z Value: " z " milli Gs");
}
```

4. Comunicación entre el Arduino y la FPGA.

```
#include "Wire.h"
#include "Arduino.h"

void setup() {
   Serial.begin(115200);
   LDOs_I2C_config();
   Wire1.begin();
}

void loop() {
  uint8_t FPGA_received=0;
  uint8_t bytes_rcv = 0;

  bytes_rcv = Wire1.requestFrom(0x30, 1, 0x81, 1, true);
  if (bytes_rcv == 1)
  {
    Serial.print("dins");
    while(Wire1.available() > 0)
      FPGA_received = Wire1.read();

  }
  Serial.println(FPGA_received);
}
```

5. Comunicación del Arduino con el sensor de temperatura LM95071

```
#include "SPI.h"
#include "Arduino.h"

void setup() {
  Serial.begin(115200);
  LDOs_I2C_config();
  SPI.begin();
  SPI.setBitOrder(MSBFIRST);
  SPI.setDataMode(SPI_MODE0);
}
```

```
void loop() {
   int16_t value;
   uint8_t bytes_rcv = 0;

   digitalWrite(13, LOW);
   byte response1 = SPI.transfer(53, 0x00, SPI_CONTINUE);
   byte response2 = SPI.transfer(53, 0x00);
   digitalWrite(13, HIGH);

   value=response1<<8;
   value = value | response2;
   value = value>>2;
   value=value*0.031125;

   Serial.println(value);
}
```

6. Comunicación entre el potenciómetro y el Arduino

```
#include "Wire.h"
#include "SPI.h"

void setup() {
  // put your setup code here, to run once:
   Serial.begin(115200);
   SPI.begin(53);
   SPI.setBitOrder(MSBFIRST);
   SPI.setDataMode(SPI_MODE0);
}

void loop() {
   int valor=29 //valor del potenciómetro
   digitalWrite(8, LOW);
   SPI.transfer(53, valor);
   digitalWrite(8, HIGH);
}
```

7. Código completo del Controlino, subrayado en verde las partes añadidas:

```
/*
 controlino.cpp - Library for controling an Arduino using the USB
 Created by Joel Koenka, April 2014
 Released under GPLv3

 Controlino let's a user control the Arduino pins by issuing simple serial commands such as "Read" "Write" etc.
 It was originally written to be used with Instrumentino, the open-source GUI platform for experimental settings,
 but can also be used for other purposes.
 For Instrumentino, see:
 http://www.sciencedirect.com/science/article/pii/S0010465514002112  - Release article
 https://pypi.python.org/pypi/instrumentino/1.0          - Package in PyPi
 https://github.com/yoelk/instrumentino              - Code in GitHub

 Modifiers (THIS IS IMPORTANT !!! PLEASE READ !!!)
 ================================================
 - Library support:
   Controlino gradually grows to include support for more and more Arduino libraries (such as PID, SoftwareSerial, etc.)
   Since not everyone needs to use all of the libraries, a set of #define statements are used to include/exclude them.

 - Arduino board:
   Controlino can run on any Arduino, but you need to tell it which one!
*/

/* ------------------------------------------------------------
Dear user (1):
Here you should specify which Arduino libraries you want to use.
Please comment/uncomment the appropriate define statements
------------------------------------------------------------ */
//#define USE_SOFTWARE_SERIAL
//#define USE_PID
#define USE_WIRE
#define USE_SPI

/* ------------------------------------------------------------
Dear user (2):
Here you should choose the Arduino Board for which you'll
compile Controlino. Only one model should be used (uncommented)
------------------------------------------------------------ */
```

```
//#define ARDUINO_BOARD_UNO
//#define ARDUINO_BOARD_LEONARDO
//#define ARDUINO_BOARD_DUE
//#define ARDUINO_BOARD_YUN
//#define ARDUINO_BOARD_TRE
//#define ARDUINO_BOARD_ZERO
//#define ARDUINO_BOARD_MICRO
//#define ARDUINO_BOARD_ESPLORA
//#define ARDUINO_BOARD_MEGA_ADK
#define ARDUINO_BOARD_MEGA_2560
//#define ARDUINO_BOARD_ETHERNET
//#define ARDUINO_BOARD_ROBOT
//#define ARDUINO_BOARD_MINI
//#define ARDUINO_BOARD_NANO
//#define ARDUINO_BOARD_LILYPAD
//#define ARDUINO_BOARD_LILYPAD_SIMPLE
//#define ARDUINO_BOARD_LILYPAD_SIMPLE_SNAP
//#define ARDUINO_BOARD_LILYPAD_USB
//#define ARDUINO_BOARD_PRO
//#define ARDUINO_BOARD_PRO_MINI
//#define ARDUINO_BOARD_FIO


/* ------------------------------------------------------------
From here down, you shouldn't touch anything (unless you know
what you're doing)
------------------------------------------------------------ */
#include "Arduino.h"
#include "string.h"
#include "HardwareSerial.h"
#include <LIS331.h>
LIS331 lis;
#include <LIS3MDL.h>
LIS3MDL mag;
#ifdef USE_WIRE
#include "Wire.h"
#endif
#ifdef USE_SPI
#include "SPI.h"
#endif

// Default values, to be overridden later
#define HARD_SER_MAX_PORTS  0
#define SOFT_SER_MAX_PORTS  0


// Arduino Uno
#ifdef ARDUINO_BOARD_UNO
  #define DIGI_PINS 14
#endif

// Arduino Leonardo
#ifdef ARDUINO_BOARD_LEONARDO
  #define DIGI_PINS 20

  #define HARD_SER_MAX_PORTS  1
  extern HardwareSerial Serial1;
#endif

// Arduino Due
#ifdef ARDUINO_BOARD_DUE
  #define DIGI_PINS 54
```

```
  #define HARD_SER_MAX_PORTS  3
  extern USARTClass Serial1;
  extern USARTClass Serial2;
  extern USARTClass Serial3;
#endif

// Arduino Yun
#ifdef ARDUINO_BOARD_YUN
  #define DIGI_PINS 20
#endif

// Arduino Tre
#ifdef ARDUINO_BOARD_TRE
  #define DIGI_PINS 14
#endif

// Arduino Zero
#ifdef ARDUINO_BOARD_ZERO
  #define DIGI_PINS 14
#endif

// Arduino Micro
#ifdef ARDUINO_BOARD_MICRO
  #define DIGI_PINS 20
#endif

// Arduino Mega ADK
#ifdef ARDUINO_BOARD_MEGA_ADK
  #define DIGI_PINS 54

  #define HARD_SER_MAX_PORTS  3
  extern HardwareSerial Serial1;
  extern HardwareSerial Serial2;
  extern HardwareSerial Serial3;
#endif

// Arduino Mega 2560
#ifdef ARDUINO_BOARD_MEGA_2560
  #define DIGI_PINS 54

  #define HARD_SER_MAX_PORTS  3
  extern HardwareSerial Serial1;
  extern HardwareSerial Serial2;
  extern HardwareSerial Serial3;
#endif

// Arduino Ethernet
#ifdef ARDUINO_BOARD_ETHERNET
  #define DIGI_PINS 14
#endif

// Arduino Nano
#ifdef ARDUINO_BOARD_NANO
  #define DIGI_PINS 14
#endif

// Arduino Lilypad
#ifdef ARDUINO_BOARD_LILIPAD
  #define DIGI_PINS 14
#endif
```

```
// Arduino Lilypad Simple
#ifdef ARDUINO_BOARD_LILYPAD_SIMPLE
  #define DIGI_PINS 9
#endif

// Arduino Lilypad Simple Snap
#ifdef ARDUINO_BOARD_LILYPAD_SIMPLE_SNAP
  #define DIGI_PINS 9
#endif

// Arduino Lilypad USB
#ifdef ARDUINO_BOARD_LILYPAD_USB
  #define DIGI_PINS 9
#endif

// Arduino Pro
#ifdef ARDUINO_BOARD_PRO
  #define DIGI_PINS 14
#endif

// Arduino Pro Mini
#ifdef ARDUINO_BOARD_PRO_MINI
  #define DIGI_PINS 13
#endif

// Arduino Fio
#ifdef ARDUINO_BOARD_FIO
  #define DIGI_PINS 14
#endif


// ---------------------------------------------------------
// Arduino libraries support
// ---------------------------------------------------------
#ifdef __cplusplus
extern "C" {
#endif
void loop();
void setup();
#ifdef __cplusplus
} // extern "C"
#endif

// Extra Hardware Serial support
#if HARD_SER_MAX_PORTS > 0
  // Descriptors for hardware serial
  HardwareSerial* hardSerHandler[HARD_SER_MAX_PORTS];
#endif

// SoftwareSerial library
#ifdef USE_SOFTWARE_SERIAL
  #include "SoftwareSerial.h"

  #define SOFT_SER_MSG_SIZE 100
  #define SOFT_SER_MAX_PORTS  4

  // software serial descriptor
  typedef struct {
    SoftwareSerial* handler;
    char txMsg[SOFT_SER_MSG_SIZE];
    char rxMsg[SOFT_SER_MSG_SIZE];
    int txMsgLen;
```

```
    int rxMsgLen;
  } SoftSerialDesc;

  // Descriptors for software serial
  SoftSerialDesc softSerDescs[SOFT_SER_MAX_PORTS];
#endif

// PID library
#ifdef USE_PID
  #include "PID_v1.h"

  // PID
  #define PID_RELAY_MAX_VARS  4

  // PID-relay variable descriptor
  typedef struct {
    PID* handler;
    int pinAnalIn;
    int pinDigiOut;
    double inputVar;
    double outputVar;
    unsigned long windowSize;
    unsigned long windowStartTime;
    double setPoint;
    boolean isOn;
  } PidRelayDesc;

  // Descriptors for PID controlled variables
  PidRelayDesc pidRelayDescs[PID_RELAY_MAX_VARS];

#endif

// ----------------------------------------------------------
// Definitions
// ----------------------------------------------------------

// arduino definitions
#define ANAL_OUT_VAL_MAX  255

// serial communication with user (usually with Instrumentino)
#ifdef ARDUINO_BOARD_DUE
extern UARTClass Serial;
#else
extern HardwareSerial Serial;
#endif

#define SERIAL0_BAUD    115200
#define RX_BUFF_SIZE    200
#define ARGV_MAX      30

//Valor con el que se inicializara el magnetómetro
#define ScaleLIS3MDL 0x00   //0x00 -----> +/-4 G     6842 LSB/G
                            //0x20 -----> +/-8 G     3421 LSB/G
                            //0x40 -----> +/-12 G    2281 LSB/G
                            //0x60 -----> +/-16 G    1711 LSB/G
//Valor con el que se inicializara el potenciómetro
#define ValPot 0xFA         //0x00 to 0xFF

// ----------------------------------------------------------
// Globals
// ----------------------------------------------------------
```

```
char doneString[5] = "done";

// Buffer to keep incoming commands and a pointer to it
char msg[RX_BUFF_SIZE];
char *pMsg;

// Pin blinking
boolean startBlinking = false;
int blinkingPin;
unsigned long blinkLastChangeMs;
unsigned long blinkingDelayMs;

// ------------------------------------------------------------
// Command functions - These functions are called when their
// respective command was issued by the user
// ------------------------------------------------------------

/***
 * Set [pin number] [in | out]
 *
 * Set a digital pin mode
 */
void cmdSet(char **argV) {
  int pin = strtol(argV[1], NULL, 10);
  char* mode = argV[2];

  if (strcasecmp(mode, "in") == 0) {
    pinMode(pin, INPUT);
  } else if (strcasecmp(mode, "out") == 0) {
    pinMode(pin, OUTPUT);
  } else {
    return;
  }
}

/***
 * Reset
 *
 * Reset all digital pins to INPUT
 */
void cmdReset() {
  for (int i = 0; i < DIGI_PINS; i++) {
    pinMode(i, INPUT);
  }
}

//comentado, daba conflicto con la lectura del sensor de temperatura
/***
 * BlinkPin
 *
 * Start blinking a pin (e.g pin 13 with the LED)
 */
 /*
void cmdBlinkPin(char **argV) {
  blinkingPin = strtol(argV[1], NULL, 10);
  blinkingDelayMs = strtol(argV[2], NULL, 10);

  pinMode(blinkingPin, OUTPUT);
  blinkLastChangeMs = millis();
  startBlinking = true;
}*/
```

```
/***
 * Read [pin1] [pin2] ...
 *
 * Read pin values
 * Pins are given in the following way: A0 A1 ... for analog pins
 *                 D0 D1 ... for digital pins
 * Answer is: val1 val2 ...
 */
void cmdRead(int argC, char **argV) {
  char pinType[2];
  int pin;
  int16_t x,y,z, value;
  float value2;
  uint8_t bytes_rcv = 0;

  for (int i = 1; i <= argC; i++) {
    pinType[0] = argV[i][0];
    pinType[1] = NULL;
    pin = strtol(&(argV[i][1]), NULL, 10);

    if (pin==22||pin==33||pin==23){   //El número de pines se puede cambiar mientras también se cambie en el archivo de configuración del Instrumentino
      mag.read();                     //Se hace una lectura de los 3 ejes del magnetómetro
    }
    if (pin == 22) {
       x=mag.m.x;
       value = x;
    } else if (pin == 33) {
       y=mag.m.y;
       value=y;
    } else if (pin == 23) {
       z=mag.m.z;
       value=z;

    }else if (pin == 62) {          //Lectura del sensor de temperatura
      noInterrupts();               //Es necesario multiplicar por 0,0031125 (LSB) en el Instrumentino para obtener el valor en ºC
      digitalWrite(13, LOW);
      byte response1 = SPI.transfer(53, 0x00, SPI_CONTINUE);
      byte response2 = SPI.transfer(53, 0x00);
      digitalWrite(13, HIGH);

      value=response1<<8;
      value = value | response2;
      value = value>>2;
      interrupts();
    } else if (pin == 61){          //Lectura de la FPGA, el registro 0x81 con la dirección 0x00
      noInterrupts();
      bytes_rcv = Wire1.requestFrom(0x30, 1, 0x81, 1, true);
      if (bytes_rcv == 1){
        while(Wire1.available() > 0) value = Wire1.read();
      }
      interrupts();
    }else if (strcasecmp(pinType, "D") == 0) {    //En caso de no realizar recibir ninguno de los otros valores de pin
      value = digitalRead(pin);                   //Se hace la lectura de pin analogico o digital convencional
    } else if (strcasecmp(pinType, "A") == 0) {
      value = analogRead(pin);
    } else {
      return;
    }
    // Add read values to answer string
    Serial.print(value);
    if (i < argC) {
```

```
      Serial.print(' ');
    }

  }
}

/***
 * Write [pin number] [digi | anal] [value]
 *
 * Write a value to a pin
 */
void cmdWrite(char **argV) {
  int pin = strtol(argV[1], NULL, 10);
  char* type = argV[2];
  int value = strtol(argV[3], NULL, 10);

  if (value==4){                          //Rango y sensibilidad del sensor magnético
  mag.writeReg(LIS3MDL::CTRL_REG2,0x00);      //Según se reciba 4,8,12 o 16 se realiza el cambio oportuno
  }else if (value==8){
  mag.writeReg(LIS3MDL::CTRL_REG2,0x20);
  }else if (value==12){
  mag.writeReg(LIS3MDL::CTRL_REG2,0x40);
  }else if (value==16){
  mag.writeReg(LIS3MDL::CTRL_REG2,0x60);
  }else if (strcasecmp(type, "digi") == 0) {  //en caso de recibir un pin estándar el programa funciona de forma convencional
    if (value == 0) {
      digitalWrite(pin, LOW);
    } else {
      digitalWrite(pin, HIGH);
    }
  } else if (strcasecmp(type, "anal") == 0) {
    analogWrite(pin, max(0, min(ANAL_OUT_VAL_MAX, value)));
  } else {
    return;
  }
}

/***
 * SetPwmFreq [pin number] [divider]
 *
 * Change the PWM frequency by changing the clock divider
 * This should be done carefully, as the clocks may have other effects on the system.
 * Specifically, pins 5,6 are controlled by timer0, which is also in charge for the delay() function.
 *
 * The divider can get: 1,8,64,256,1024     for pins 5,6,9,10;
 *           1,8,32,64,128,256,1024  for pins 3,11
 */
void cmdSetPwmFreq(char **argV) {
  int pin = strtol(argV[1], NULL, 10);
  int divider = strtol(argV[2], NULL, 10);

  byte mode;
  if(pin == 5 || pin == 6 || pin == 9 || pin == 10) {
    switch(divider) {
      case 1: mode = 0x01; break; //    5,6: 62500 Hz | 9,10: 31250 Hz
      case 8: mode = 0x02; break; //    5,6: 7812.5 Hz  | 9,10: 3906.3 Hz
      case 64: mode = 0x03; break; //   5,6: 976.6 Hz | 9,10: 488.3 Hz
      case 256: mode = 0x04; break; //  5,6: 244.1 Hz | 9,10: 122 Hz
      case 1024: mode = 0x05; break;//  5,6: 61 Hz    | 9,10: 30.5 Hz
      default: return;
    }
    if(pin == 5 || pin == 6) {
```

```
#if defined(TCCR0B)
      TCCR0B = (TCCR0B & 0b11111000) | mode;
#endif
    } else {
#if defined(TCCR1B)
      TCCR1B = (TCCR1B & 0b11111000) | mode;
#endif
    }
  } else if(pin == 3 || pin == 11) {
    switch(divider) {
      case 1: mode = 0x01; break; //    31250 Hz
      case 8: mode = 0x02; break; //    3906.3 Hz
      case 32: mode = 0x03; break; //   976.6 Hz
      case 64: mode = 0x04; break; //   488.3 Hz
      case 128: mode = 0x05; break; //  244.1 Hz
      case 256: mode = 0x06; break; //  122 Hz
      case 1024: mode = 0x7; break; //  30.5 Hz
      default: return;
    }
#if defined(TCCR2B)
    TCCR2B = (TCCR2B & 0b11111000) | mode;
#endif
  }
}

#ifdef USE_PID
/***
 * PidRelayCreate [pidVar] [pinAnalIn] [pinDigiOut] [windowSize] [Kp] [Ki] [Kd]
 *
 * Create a PID variable that controls a relay. window size is in ms.
 * See more information: http://playground.arduino.cc/Code/PIDLibraryRelayOutputExample
 */
void cmdPidRelayCreate(char **argV) {
  int pidVar = strtol(argV[1], NULL, 10);
  int pinAnalIn = strtol(argV[2], NULL, 10);
  int pinDigiOut = strtol(argV[3], NULL, 10);
  int windowSize = strtol(argV[4], NULL, 10);
  double kp = atof(argV[5]);
  double ki = atof(argV[6]);
  double kd = atof(argV[7]);

  if (pidVar < 1 || pidVar > PID_RELAY_MAX_VARS) {
    return;
  }

  // Init the PID variable
  PidRelayDesc* pidDesc = &pidRelayDescs[pidVar-1];
  pidDesc->pinAnalIn = pinAnalIn;
  pidDesc->pinDigiOut = pinDigiOut;
  pidDesc->windowSize = windowSize;
  pidDesc->handler = new PID(&pidDesc->inputVar, &pidDesc->outputVar, &pidDesc->setPoint, kp, ki, kd, DIRECT);
  pidDesc->handler->SetOutputLimits(0, pidDesc->windowSize);
  pidDesc->isOn = false;
}

/***
 * PidRelayTune [Kp] [Ki] [Kd]
 *
 * Set the PID tuning parameters
 */
void cmdPidRelayTune(char **argV) {
  int pidVar = strtol(argV[1], NULL, 10);
```

```
  double kp = atof(argV[2]);
  double ki = atof(argV[3]);
  double kd = atof(argV[4]);

  if (pidVar < 1 || pidVar > PID_RELAY_MAX_VARS) {
    return;
  }

  PidRelayDesc* pidDesc = &pidRelayDescs[pidVar-1];
  pidDesc->handler->SetTunings(kp, ki, kd);
}

/***
 * PidRelaySet [pidVar] [setpoint]
 *
 * Start controlling a relay using a PID variable
 */
void cmdPidRelaySet(char **argV) {
  int pidVar = strtol(argV[1], NULL, 10);
  int setPoint = strtol(argV[2], NULL, 10);

  if (pidVar < 1 || pidVar > PID_RELAY_MAX_VARS) {
    return;
  }

  PidRelayDesc* pidDesc = &pidRelayDescs[pidVar-1];
  pidDesc->setPoint = setPoint;
}

/***
 * PidRelayEnable [pidVar] [0 | 1]
 *
 * Start/Stop the control loop
 */
void cmdPidRelayEnable(char **argV) {
  int pidVar = strtol(argV[1], NULL, 10);
  int enable = strtol(argV[2], NULL, 10);

  if (pidVar < 1 || pidVar > PID_RELAY_MAX_VARS) {
    return;
  }

  PidRelayDesc* pidDesc = &pidRelayDescs[pidVar-1];
  pidDesc->windowStartTime = millis();

  // turn the PID on/off
  if (enable != 0) {
    pidDesc->isOn = true;
    pidDesc->handler->SetMode(AUTOMATIC);
  } else {
    pidDesc->isOn = false;
    pidDesc->handler->SetMode(MANUAL);
    digitalWrite(pidDesc->pinDigiOut, LOW);
  }
}
#endif

/***
 * HardSerConnect [baudrate] [port]
 *
 * Initiate a software serial connection. The rx-pin should have external interrupts
 */
```

```
void cmdHardSerConnect(char **argV) {
#if HARD_SER_MAX_PORTS > 0
  int baudrate = strtol(argV[1], NULL, 10);
  int currPort = strtol(argV[2], NULL, 10);

  if (currPort < 1 || currPort > HARD_SER_MAX_PORTS) {
    return;
  }

  // begin serial communication
  hardSerHandler[currPort-1]->begin(baudrate);
#endif
}

/***
 * SoftSerConnect [rx-pin number] [tx-pin number] [baudrate] [port]
 *
 * Initiate a software serial connection. The rx-pin should have external interrupts
 */
#ifdef USE_SOFTWARE_SERIAL
void cmdSoftSerConnect(char **argV) {
  int pinIn = strtol(argV[1], NULL, 10);
  int pinOut = strtol(argV[2], NULL, 10);
  int baudrate = strtol(argV[3], NULL, 10);
  int currPort = strtol(argV[4], NULL, 10);

  if (currPort < 1 || currPort > SOFT_SER_MAX_PORTS) {
    return;
  }

  // init softSerial struct
  softSerDescs[currPort-1].rxMsgLen = 0;
  softSerDescs[currPort-1].txMsgLen = 0;
  softSerDescs[currPort-1].handler = new SoftwareSerial(pinIn, pinOut, false);
  softSerDescs[currPort-1].handler->begin(baudrate);
}
#endif

/***
 * SerSend [hard | soft] [port]
 *
 * After this command, each character sent is mirrored to the chosen serial
 * port until the NULL character (0x00) is sent (also mirrored)
 */
void cmdSerSend(char **argV) {
  boolean isSoftSerial = (strcasecmp(argV[1], "soft") == 0);
  int currPort = strtol(argV[2], NULL, 10);
  Serial.println(doneString);

  if (currPort < 1 || currPort > ((isSoftSerial)? SOFT_SER_MAX_PORTS : HARD_SER_MAX_PORTS)) {
    return;
  }
  if (isSoftSerial) {
#ifdef USE_SOFTWARE_SERIAL
    softSerDescs[currPort-1].txMsgLen = 0;
#endif
  }

  // mirror the hardware serial and the software serial
  while (true) {
    if (Serial.available()) {
      char c = Serial.read();
```

```
    if (isSoftSerial) {
#ifdef USE_SOFTWARE_SERIAL
      softSerDescs[currPort-1].txMsg[softSerDescs[currPort-1].txMsgLen++] = c;
#endif
    } else {
#if HARD_SER_MAX_PORTS > 0
      hardSerHandler[currPort-1]->write(c);
#else
      return;
#endif
    }

    if (c == '\0') {
      // acknowledge
      Serial.println(doneString);
      delay(10);
#ifdef USE_SOFTWARE_SERIAL
      if (isSoftSerial) {
        // send the message, and remember the answer
        for (int i = 0; i < softSerDescs[currPort-1].txMsgLen; i++) {
          softSerDescs[currPort-1].handler->write(softSerDescs[currPort-1].txMsg[i]);
        }
        softSerDescs[currPort-1].rxMsgLen = 0;
        while (!Serial.available()) {
          if (softSerDescs[currPort-1].handler->available() && softSerDescs[currPort-1].rxMsgLen < SOFT_SER_MSG_SIZE)
{
                                                    softSerDescs[currPort-1].rxMsg[softSerDescs[currPort-1].rxMsgLen++]    =
softSerDescs[currPort-1].handler->read();
          }
        }
      }
#endif
      return;
    }
  }
 }
}

/***
 * SerReceive [hard | soft] [port]
 *
 * Empty the RX buffer of a serial port to the control serial port
 */
void cmdSerReceive(char **argV) {
  boolean isSoftSerial = (strcasecmp(argV[1], "soft") == 0);
  int currPort = strtol(argV[2], NULL, 10);

  if (currPort < 1 || currPort > (isSoftSerial)? SOFT_SER_MAX_PORTS : HARD_SER_MAX_PORTS) {
    return;
  }

  if (isSoftSerial) {
#ifdef USE_SOFTWARE_SERIAL
    for (int i = 0; i < softSerDescs[currPort-1].rxMsgLen && i < SOFT_SER_MSG_SIZE; i++) {
      Serial.write(softSerDescs[currPort-1].rxMsg[i]);
    }
#endif
  } else {
#if HARD_SER_MAX_PORTS > 0
    while (hardSerHandler[currPort-1]->available()) {
      Serial.write(hardSerHandler[currPort-1]->read());
    }
```

```
#endif
  }
}

/***
 * I2cWrite [address] [val1] [val2] ...
 *
 * Write a series of values to the I2C bus
 */
#ifdef USE_WIRE
void cmdI2cWrite(int argC, char **argV) {
  int address = strtol(argV[1], NULL, 10);

  Wire.beginTransmission(address);
  for (int i = 2; i <= argC-1; i++) {
    Wire.write(strtol(argV[i], NULL, 10));
  }
  Wire.endTransmission();
}
#endif

/***
 * SpiWrite [cs_pin] [val1] [val2] ...
 *
 * Write a series of values to the SPI bus
 */
#ifdef USE_SPI
void cmdSpiWrite(int argC, char **argV) {
  int cs_pin = strtol(argV[1], NULL, 10);

  noInterrupts();
  #ifdef ARDUINO_BOARD_DUE              //En caso de usarse el Arduino Due la función no es igual que en el Uno
     digitalWrite(8, LOW);              //Ajuste del potenciómetro
     for (int i = 2; i <= argC-1; i++) {
       SPI.transfer(cs_pin, strtol(argV[i], NULL, 10));
     }
     digitalWrite(8, HIGH);
  #else

  digitalWrite(cs_pin, LOW);
  for (int i = 2; i <= argC-1; i++) {
    SPI.transfer(strtol(argV[i], NULL, 10));
  }
  digitalWrite(cs_pin, HIGH);
  interrupts();
}
#endif
}

// ----------------------------------------------------------------
// Main functions
// ----------------------------------------------------------------

/***
 * The setup function is called once at startup of the sketch
 */
void setup() {
  Serial.begin(SERIAL0_BAUD);
  pMsg = msg;
     pinMode(13, OUTPUT);  //pin cs del sensor de temperatura configurado como salida
     pinMode(8, OUTPUT);   //pin cs del potenciómetro configurado como salida
     digitalWrite(13, HIGH);   //pin 13 a nivel alto
```

```
    digitalWrite(8,HIGH);     //pin 8 a nivel alto

#ifdef USE_WIRE
  Wire.begin();
  Wire1.begin();
#endif

   LDOs_I2C_config();  //configuramos la LDO la función esta implementada en otra ventana
   SPI.begin();
   SPI.setBitOrder(MSBFIRST);
   SPI.setDataMode(SPI_MODE0);
    //SPI.setClockDivider(16);  //En caso de querer modificar la velocidad del bus SPI

   if (!mag.init()){  //Iniciación del magnetómetro
   while (1);
   }

  mag.enableDefault();
     mag.writeReg(LIS3MDL::CTRL_REG2,ScaleLIS3MDL);          //Configuración de la sensibilidad y resolución del
magnetómetro

   digitalWrite(8, LOW);
   SPI.transfer(53, ValPot);                    //Iniciación del potenciómetro con el valor deseado
   digitalWrite(8, HIGH);


/***
 * The loop function is called in an endless loop
 */
void loop() {
  char c, argC;
  char *argV[ARGV_MAX];
  int i, pin;
  unsigned long curMs;

  // comentado, daba conflicto con la lectura del sensor de temperatura
  /*if (startBlinking == true) {
    curMs = millis();
    if (curMs > blinkLastChangeMs + blinkingDelayMs) {
      blinkLastChangeMs = curMs;
      if (digitalRead(blinkingPin) == HIGH) {
        digitalWrite(blinkingPin, LOW);
      } else {
        digitalWrite(blinkingPin, HIGH);
      }
    }
  }*/

#ifdef USE_PID
  // Take care PID-relay variables
  for (i = 0; i < PID_RELAY_MAX_VARS; i++) {
    if (pidRelayDescs[i].isOn) {
      pidRelayDescs[i].inputVar = analogRead(pidRelayDescs[i].pinAnalIn);
      pidRelayDescs[i].handler->Compute();

      // turn relay on/off according to the PID output
      curMs = millis();
      if (curMs - pidRelayDescs[i].windowStartTime > pidRelayDescs[i].windowSize) {
        //time to shift the Relay Window
        pidRelayDescs[i].windowStartTime += pidRelayDescs[i].windowSize;
      }
      if (pidRelayDescs[i].outputVar > curMs - pidRelayDescs[i].windowStartTime) {
```

```
        digitalWrite(pidRelayDescs[i].pinDigiOut, HIGH);
      }
      else {
        digitalWrite(pidRelayDescs[i].pinDigiOut, LOW);
      }
    }
  }
#endif

  // Read characters from the control serial port and act upon them
  if (Serial.available()) {
    c = Serial.read();
    switch (c) {
    case '\n':
      break;
    case '\r':
      // end the string and init pMsg
      Serial.println("");
      *(pMsg++) = NULL;
      pMsg = msg;
      // parse the command line statement and break it up into space-delimited
      // strings. the array of strings will be saved in the argV array.
      i = 0;
      argV[i] = strtok(msg, " ");

      do {
        argV[++i] = strtok(NULL, " ");
      } while ((i < ARGV_MAX) && (argV[i] != NULL));

      // save off the number of arguments
      argC = i;
      pin = strtol(argV[1], NULL, 10);

      if (strcasecmp(argV[0], "Set") == 0) {
        cmdSet(argV);
      } else if (strcasecmp(argV[0], "Reset") == 0) {
        cmdReset();
      /*} else if (strcasecmp(argV[0], "BlinkPin") == 0) {  //Comentado, daba conflicto con la lectura del sensor de temperatura
        cmdBlinkPin(argV);*/
      } else if (strcasecmp(argV[0], "Read") == 0) {
        cmdRead(argC, argV);
      } else if (strcasecmp(argV[0], "Write") == 0) {
        cmdWrite(argV);
      } else if (strcasecmp(argV[0], "SetPwmFreq") == 0) {
        cmdSetPwmFreq(argV);
#ifdef USE_PID
      } else if (strcasecmp(argV[0], "PidRelayCreate") == 0) {
        cmdPidRelayCreate(argV);
      } else if (strcasecmp(argV[0], "PidRelaySet") == 0) {
        cmdPidRelaySet(argV);
      } else if (strcasecmp(argV[0], "PidRelayTune") == 0) {
        cmdPidRelayTune(argV);
      } else if (strcasecmp(argV[0], "PidRelayEnable") == 0) {
        cmdPidRelayEnable(argV);
#endif
      } else if (strcasecmp(argV[0], "HardSerConnect") == 0) {
        cmdHardSerConnect(argV);
#ifdef USE_SOFTWARE_SERIAL
      } else if (strcasecmp(argV[0], "SoftSerConnect") == 0) {
        cmdSoftSerConnect(argV);
#endif
      } else if (strcasecmp(argV[0], "SerSend") == 0) {
```

```
        cmdSerSend(argV);
      } else if (strcasecmp(argV[0], "SerReceive") == 0) {
        cmdSerReceive(argV);
#ifdef USE_WIRE
      } else if (strcasecmp(argV[0], "I2cWrite") == 0) {
        cmdI2cWrite(argC, argV);
#endif
#ifdef USE_SPI
      }  else if (strcasecmp(argV[0], "SpiRead") == 0) {
        cmdSpiRead(argC, argV);
#endif
      } else {
        // Wrong command
        return;
      }

      // Acknowledge the command
      Serial.println(doneString);
      break;
    default:
      // Record the received character
      if (isprint(c) && pMsg < msg + sizeof(msg)) {
        *(pMsg++) = c;
      }
      break;
    }
  }
}
```

Anexo B

Código Instrumentino

1. Clase usada para el sensor magnético

```
class SysVarAnalogArduinoMag4(SysVarAnalog):

    def __init__(self, name, range, pinAnalIn, pinPwmOut=None,
    SetPolarityPositiveFunc=None, GetPolarityPositiveFunc=None, compName='', helpLine='',
    units='', PreSetFunc=None, highFreqPWM=False, pinOutVoltsMax=5, pinInVoltsMax=5,
    pinOutVoltsMin=0, pinInVoltsMin=0, PostGetFunc=None, I2cDac=None, sen_mag=6842):
    showEditBox = (pinPwmOut != None) or (PreSetFunc != None) or (I2cDac != None)
        SysVarAnalog.__init__(self, name, range, Arduino, compName, helpLine,
        showEditBox , units, PreSetFunc, PostGetFunc)
        self.pinIn = pinAnalIn
        self.pinOut = pinPwmOut
        self.SetPolarityPositiveFunc = SetPolarityPositiveFunc
        self.GetPolarityPositiveFunc = GetPolarityPositiveFunc
        self.highFreqPWM = highFreqPWM
        self.pinOutVoltsMax = pinOutVoltsMax
        self.pinInVoltsMax = pinInVoltsMax
        self.pinOutVoltsMin = pinOutVoltsMin
        self.pinInVoltsMin = pinInVoltsMin
        self.I2cDac = I2cDac
        self.sen_mag = sen_mag

    def FirstTimeOnline(self):
        if self.pinOut != None:
            self.GetController().PinModeOut(self.pinOut)
        if self.highFreqPWM:
            self.GetController().SetHighFreqPwm(self.pinOut)

    def GetUnipolarRange(self):
        return 1
        def GetFunc(self):
                fraction = self.GetController().AnalogReadFraction(self.pinIn,
            self.pinInVoltsMax, self.pinInVoltsMin)
            sign = 1 if self.GetPolarityPositiveFunc() else -
                return sign * ((self.GetUnipolarMin() + (self.GetUnipolarRange() *
            fraction))/self.sen_mag)*100 if fraction != None else None
            def SetFunc(self, value):
            fraction = (abs(value) - self.GetUnipolarMin()) / self.GetUnipolarRange()
            if value == 12:
                self.sen_mag =2281
            if value ==16:
                self.sen_mag =1711
            if value ==8:
                self.sen_mag =3421
            if value ==4:
                self.sen_mag =6842
```

```
if self.pinOut != None:
        self.GetController().AnalogWriteFraction(self.pinOut, fraction,
        self.pinOutVoltsMax, self.pinOutVoltsMin)
elif self.I2cDac != None:
        minV = self.pinOutVoltsMin
        maxV = self.pinOutVoltsMax
self.I2cDac.WriteFraction((minV + (maxV - minV) * fraction) / 5,
self.GetController())
```

2. Clase que implementa el componente potenciometro

```
from __future__ import division

class ArduinoPote(object):
        def __init__(self, dacBits):
                def WriteFraction(self, fraction, controller):
                pass

class Potenciometro(ArduinoPote):
        def __init__(self, cs_pin, channel):
                self.dacBits = 8
                self.cs_pin = cs_pin
                self.channel = channel
                super(Potenciometro, self).__init__(self.dacBits)

        def WriteFraction(self, fraction, controller):
                controller.SpiWrite(self.cs_pin, (fraction))
```

3. Archivo de configuración final

```
from __future__ import division
from instrumentino import Instrument
from instrumentino import cfg
from instrumentino.action import SysAction, SysActionParamTime, SysActionParamFloat, SysActionParamInt
from instrumentino.controllers.arduino import SysVarDigitalArduino, SysVarAnalogArduinoUnipolar,
SysVarAnalogArduinoUnipolarSPI,SysVarAnalogArduinoUnipolarTemp,SysVarAnalogArduinoUnipolarMag4,
SysVarAnalogArduinoTemp
from instrumentino.controllers.arduino.pins import DigitalPins, AnalogPins
from instrumentino.controllers.arduino.dac import DacSpiMCP4922, DacI2cMAX517
'''
*** System constants
'''
# Arduino pins
```

```
pinAnalInA = 22
pinPwmOutA = 9
pinAnalInB = 33
pinPwmOutB = 9
pinAnalInC = 23
pinPwmOutC = 9
pinAnalInT = 62
pinPwmOutT = 9

spi_dac1 = DacSpiMCP4922(4, 1)
spi_dac1_anal_in = 15

sss=SysVarAnalogArduinoUnipolarMag4

'''
*** System components
'''

spi_dac_channels = AnalogPins('Temperature & Res',
(SysVarAnalogArduinoUnipolarTemp(' ',[0,255],pinAnalInT,None, units='V', I2cDac=spi_dac1),))



analPins = AnalogPins('Accelerometer', (sss('X', (0,16), pinAnalInA, pinPwmOutC, 'axis', units='uT'),
sss('Y', (0,16), pinAnalInB, pinPwmOutC, 'axis', units='uT'),
sss('Z', (0,16), pinAnalInC, pinPwmOutC, 'axis', units='uT'),))


'''
*** System actions
'''
class SysActionSetPins(SysAction):
def __init__(self):
self.seconds = SysActionParamTime()
self.pinA = SysActionParamFloat(analPins.vars['X'])
SysAction.__init__(self, 'Set pins', (self.seconds, self.pinA))
def Command(self):
analPins.vars['X'].Set(self.pinA.Get())
cfg.Sleep(self.seconds.Get())
analPins.vars['X'].Set(0)


'''
*** System
'''
class System(Instrument):
def __init__(self):
comps = (analPins,spi_dac_channels,)
actions = (SysActionSetPins(),)
name = ''
description = ''
```

```
version = '1'
Instrument.__init__(self, comps, actions, version, name, description)
'''
*** Run program
'''
if __name__ == '__main__':
# run the program
System()
```